Dr. Dobb's JOURNAL

SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

VOLUME 15

1990

Featuring the year's hottest topics: neural nets, graphics, memory management, real-time, and much more!
Also includes the complete source code from 1990.

Dr. Dobb's JOURNAL

SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

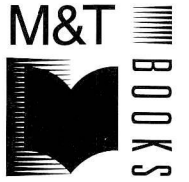
VOLUME 15

1990

Featuring the year's hottest topics: neural nets, graphics, memory management, real-time, and much more!
Also includes the complete source code from 1990.







A Publication of M&T Publishing, Inc., Redwood City, California

M&T Books
A Division of M&T Publishing, Inc.
501 Galveston Drive
Redwood City, CA 94063

© 1991 by M&T Publishing, Inc.

Printed in the United States of America

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without prior written permission from the Publisher. Contact the Publisher for information on foreign rights.

Limits of Liability and Disclaimer of Warranty
The Author and Publisher of this book have used their best efforts in preparing the book and the programs contained in it. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness.

The Author and Publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The Author and Publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Library of Congress Cataloging in Publication Data available from publisher.

ISBN 1-55851-183-0 (book/disk set)

Trademarks:
All products, names, and services are trademarks or registered trademarks of their respective companies.

94 93 92 91 4 3 2 1

About M&T Publishing and *Dr. Dobb's Journal*

M&T Publishing, Inc., was founded in 1981 as the American subsidiary of Markt & Technik A.G., the leading German publisher of computer magazines, books, and software. In addition to its active role in the international microcomputer marketplace, M&T publishes three monthly magazines: *Dr. Dobb's Journal*, *DBMS*, and *LAN Technology*. M&T also publishes an extensive line of microcomputer books and software under the imprint M&T Books.

Since its first issue in 1976, *Dr. Dobb's Journal* has maintained a unique voice among microcomputer publications. The first issue was published to provide a short-term distribution of the newly written Tiny BASIC language. Reader response was so strong that it immediately went into monthly production. *DDJ* has led the way since then by focusing on important advances in microcomputers, printing public-domain software, and fostering vital reader interaction on technical matters.

Today, *Dr. Dobb's Journal* remains the primary source of information and software tools for advanced programmers. Its early issues are still in demand, and no part of the editorial content of *DDJ* has ever gone out of print. Volumes One through Fifteen of this series contain the entire editorial content of *DDJ*'s first fifteen years and are available through M&T Books.

Through *Dr. Dobb's Journal* and its other software publications, M&T Publishing is providing microcomputer professionals and enthusiasts with the most advanced and useful software information available.

Contents

13 Editor's Preface

15 JANUARY 1990, NUMBER 160

17 Editorial — JONATHAN ERICKSON
21 Real-Time Animation — RAHNER JAMES
26 Real-Time Data Acquisition Using DMA — TOM NOLAN
32 Zen for Embedded Systems — MARTIN TRACY
36 Error Message Management — ROHAN T. DOUGLAS
38 S-Coder for Data Encryption — ROBERT B. STOUT
42 Parametric Circles — ROBERT ZIGON
43b Examining Zortech C++ 2.0 — SCOTT ROBERT LADD
48 Stalking General Protection Faults: Part I — ANDREW SCHULMAN
79 Location is Everything! — MARK R. NELSON
83 Programming Paradigms — MICHAEL SWAINE
86 C Programming — AL STEVENS
92 Structured Programming — JEFF DUNTEMANN
103 Of Interest
106 Swaine's Flames — MICHAEL SWAINE

107 FEBRUARY 1990, NUMBER 161

109 Editorial — JONATHAN ERICKSON
113 Managing Multiple Data Segments Under Microsoft Windows: Part I — TIM PATERSON and STEVE FLENNIKEN
120 Three-Dimensional Graphics Using the X Window System — MICHAEL STROYAN
125 Pick-A-Number Interfaces — BOB CANUP
127 Self-Adjusting Data Structures — ANDREW M. LIAO
135 Multiplexing Error Codes — WILLIAM J. MCMAHON
139 C_Talk/Views — NOEL BERGMAN
145 Stalking General Protection Faults: Part II — ANDREW SCHULMAN

170	Programming Risc Engines	NEAL MARGULIS
172	Programming Paradigms	MICHAEL SWAINE
175	C Programming	AL STEVENS
183	Structured Programming	JEFF DUNTEMANN
195	Of Interest	
198	Swaine's Flames	MICHAEL SWAINE

199 MARCH 1990, NUMBER 162

201	Editorial	JONATHAN ERICKSON
205	Assembly Language Lives!	MICHAEL ABRASH
212	Assembly Language Tricks of the Trade	TIM PATERSON
216	68040 Programming	STEPHEN SATCHELL
220	Homegrown Debugging - 386 Style!	AL WILLIAMS
227	Managing Multiple Data Segments Under Microsoft Windows: Part II	TIM PATERSON and STEVE FLENNIKEN
232	Object-Oriented Programming with Assembly Language	RANDALL HYDE
237	Inside Watcom C 7.0/386	ANDREW SCHULMAN
243	Mixed-Language Programming with ASM	KARL WRIGHT and RICK SCHELL
265	Programming Paradigms	MICHAEL SWAINE
269	C Programming	AL STEVENS
274	Structured Programming	JEFF DUNTEMANN
285	Of Interest	
288	Swaine's Flames	MICHAEL SWAINE

289 APRIL 1990, NUMBER 163

291	Editorial	JONATHAN ERICKSON
295	Bidirectional Associative Memory Systems in C++	ADAM BLUM
301	A Neural Network Instantiation Environment	ANDREW J. CZUCHRY, JR.
307	Untangling Neural Nets	JEANNETTE "JET" LAWRENCE
312	Implementing the Rhealstone Real-Time Benchmark	RABINDRA P. KAR
319	Bounding Box Data Compression	GLENN SEARFOSS
324	1989 DDJ Index	
328	VESA VGA Bios Extensions	BO ERICCSON
333	Cruising with Topspeed	ALEX LANE
337	Neural Networks and Image Processing	CASIMIR C. "CASEY" KLIMASAUSKAS
364	Programming Paradigms	MICHAEL SWAINE

369	C Programming	AL STEVENS
374	Structured Programming	JEFF DUNTEMANN
386	Of Interest	
389	Swaine's Flames	MICHAEL SWAINE

390 MAY 1990, NUMBER 164

392	Editorial	JONATHAN ERICKSON
396	Generation Scavenging	FRANK JACKSON
404	Dynamic Link Libraries For DOS	GARY SYCK
411	Getting A Handle on Virtual Memory	WALTER BRIGHT
416	Object Swapping	JAN BOTTORFF and JIM BOLLAND
422	A Memory Controller	ROBERT A. MOESER
428	Demystifying 16-Bit VGA	MICHAEL ABRASH
435	Multiprocessing With Smalltalk/V	KENNETH E. AYERS
441	Accessing Hardware From 80386 Protected Mode: Part I	STEPHEN FRIED
460	Programming Paradigms	MICHAEL SWAINE
464	C Programming	AL STEVENS
470	Structured Programming	JEFF DUNTEMANN
477	Encapsulating C++ Books	ANDREW SCHULMAN
479	Of Interest	
481	Swaine's Flames	MICHAEL SWAINE

483 JUNE 1990, NUMBER 165

485	Editorial	JONATHAN ERICKSON
489	The DDJ Hypertext Project	J. SCOOT JOHNSON
492	Building A Hypertext System	RICK GESSNER
498	A Self-Referential Hypertext Engine	TODD KING
501	Building An Efficient Help System	LEO NOTENBOOM and MICHAEL VOSE
506	C++ File Objects	KEVIN WEEKS
511	A Pixel Ordering Algorithm	NORTON T. ALLEN
515	Examining Instant-C	ANDREW SCHULMAN
523	Accessing Hardware From 80386 Protected Mode: Part II	STEPHEN FRIED

553 LZW Revisited — SHAWN M. REGAN
555 Programming Paradigms — MICHAEL SWAINE
560 C Programming — AL STEVENS
565 Structured Programming — JEFF DUNTEMANN
577 Of Interest
580 Swaine's Flames — MICHAEL SWAINE

581 JULY 1990, NUMBER 166

583 Editorial — JONATHAN ERICKSON
587 Super VGA Programming — CHRISTOPHER A. HOWARD
593 Circles and the Digital Differential Analyzer — TIM PATERSON
597 Improving Line Segment Clipping — VICTOR J. DUVANENKO, W. E. ROBBINS, and ONALD S. GYURCSIK
603 Drawing Character Shapes With Bézier Curves — TODD KING
607 Information Models, Views, and Controllers — ADELE GOLDBERG
613 DOS + 386 = 4 Gigabytes! — AL WILLIAMS
619 The Power in Powerbasic — BRUCE TONKIN
649 Programming Paradigms — MICHAEL SWAINE
653 C Programming — AL STEVENS
659 Structured Programming — JEFF DUNTEMANN
669 Of Interest
671 Swaine's Flames — MICHAEL SWAINE

673 AUGUST 1990, NUMBER 167

675 Editorial — JONATHAN ERICKSON
678 Porting C Programs to 80386 Protected Mode — WILLIAM F. DUDLEY, JR.
682 Encapsulating C Memory Allocation — JIM SCHIMANDLE
689 AWK as a C Code Generator — WAHHAB BALDWIN
695 Implementing Bicubic Splines — RAYMOND G. LAUZZANA and DENISE E. M. PENROSE
704 Extending Printf() — JIM MISCHEL
709 Parallel Extensions to C — GRAHAM K. ELLIS
716 Debugging Memory Allocation Errors — LAWRENCE D. SPENCER
720 Optimizing with Microsoft C 6.0 — SCOTT ROBERT LADD
727 Collections in Turbo C++ — BRUCE ECKEL

756	Handling OS/2 Error Codes	NICO MAK
758	Programming Paradigms	MICHAEL SWAINE
765	C Programming	AL STEVENS
771	Structured Programming	JEFF DUNTEMANN
786	Of Interest	
788	Swaine's Flames	MICHAEL SWAINE

789 SEPTEMBER 1990, NUMBER 168

791	Editorial	JONATHAN ERICKSON
795	Making the Move to Modula-2	J. V. AUPING and J. C. JOHNSTON
801	Porting Fortran Programs From Minis to PCs	JOHN L. BRADBERRY
805	Persistent Objects in Turbo Pascal	SCOTT ROBERT LADD
809	Fast Search	LEON CAMPISE
813	A Generic One-Pass Assembler	WILLIAM E. IVES
818	Inside Object Professional	GARY ENTSMINGER
823	Kermit For OS/2: Part I	BRIAN R. ANDERSON
857	Programming Paradigms	MICHAEL SWAINE
862	C Programming	AL STEVENS
868	Structured Programming	JEFF DUNTEMANN
875	Programmer's Bookshelf	ANDREW SCHULMAN
880	Ray Tracing	DANIEL LYKE
886	Of Interest	
888	Swaine's Flames	MICHAEL SWAINE

889 OCTOBER 1990, NUMBER 169

891	Editorial	JONATHAN ERICKSON
895	Roll Your Own DOS Extender: Part I	AL WILLIAMS
900	Opening OS/2's Backdoor	ANDREW SCHULMAN
906	Closing DOS's Backdoor	JOHN SWITZER
910	RAM Disk Driver for UNIX	JEFF REAGEN
915	Optimal Determination of Object Extents	VICTOR J. DUVANENKO, RONALD GYURCSIK, and W. E. ROBBINS
917	Unraveling Optimization in Microsoft C 6.0	BRUCE D. SCHATZMAN

924	Kermit for OS/2: Part II	BRIAN R. ANDERSON
955	Programming Paradigms	MICHAEL SWAINE
959	C Programming	AL STEVENS
964	Structured Programming	JEFF DUNTEMANN
969	Programmer's Bookshelf	RAY DUNCAN
976	Implementing Cordic Algorithms	PITTS JARVIS
982	Of Interest	
984	Swaine's Flames	MICHAEL SWAINE

985 NOVEMBER 1990, NUMBER 170

987	Editorial	JONATHAN ERICKSON
991	Roll Your Own Object-Oriented Language	MICHAEL FLOYD
994	An Existential Dictionary	EDWIN T. FLOYD
1000	Object-Oriented Debugging	SIMON TOOKE
1004	Ctrace: A Message Logging Class	WILLIAM D. CRAMER
1009	Software Patents	THE LEAGUE FOR PROGRAMMING FREEDOM
1019	Roll Your Own DOS Extender: Part II	AL WILLIAMS
1026	Programmer Tools for Actor 3.0	MARTY FRANZ
1030	Windows 3.0 Application Development	WALTER KNOWLES
1063	Programming Paradigms	MICHAEL SWAINE
1067	C Programming	AL STEVENS
1072	Structured Programming	JEFF DUNTEMANN
1075	Programmer's Bookshelf	ANDREW SCHULMAN
1082	The MVC Paradigm in Smalltalk/V	KENNETH E. AYERS
1088	Of Interest	
1090	Swaine's Flames	MICHAEL SWAINE

1091 DECEMBER 1990, NUMBER 171

1093	Editorial	JONATHAN ERICKSON
1097	Controlling Background Processes Under UNIX	BARR E. BAUER
1101	Designing an OSI Test Bed	KENNETH L. CROCKER and MICHAEL T. THOMPSON
1108	The Macintosh Communications Toolbox	DON GASPAR
1112	Algebraic Codes for Error Detection and Correction	HSI-CHIU LIU
1116	Supercharging Sequential Searches	WALTER WILLIAMS
1122	Examining the Zinc Interface Library	GARY ENTSMINGER
1126	A Database System for Automating E-Mail	CHRIS OHLSEN
1154	Programming Paradigms	MICHAEL SWAINE
1158	C Programming	AL STEVENS
1164	Structured Programming	JEFF DUNTEMANN
1170	Programmer's Bookshelf	RAY DUNCAN
1179	Of Interest	
1182	Swaine's Flames	MICHAEL SWAINE
1183	Index	

Editor's Preface

The banner on the January 1990 cover of Dr. Dobb's Journal sums up what's perhaps our proudest accomplishment—15 Years of Power Programming! That's something no other magazine devoted exclusively to software development can come close to saying.

Over the past decade-and-a-half, Dr. Dobb's seems to have come a long way from the days of "computer calisthenics & orthodontia." In many ways we have, yet in others Dr. Dobb's remains unchanged. Certainly our content has changed as new languages, innovative CPUs, and up-to-date environments have come on the scene. However, our approach to these topics—and the spirit in which we do so—is the same. From Dennis Allison's Tiny BASIC in 1976 to DDJ's senior technical editor Michael Floyd's implementation of an object-oriented Prolog or Bill Ive's homegrown one-pass assembler (both in these pages), Dr. Dobb's Journal's mission has been to share the best in programming tools and techniques.

In addition to Michael's "Object Prolog" in the November issue, 1990 showcased many other important projects. In June, for instance, we presented the DDJ Hyertext Project, a fully-hypertexted electronic version of the issue to complement the paper page. The project, developed by Scott Johnson (and described in the June issue), included linked text, graphics, and source code all in electronic form on CompuServe and disk.

Al William's two-part "roll-your-own" DOS Extender article in October and November was also in the DDJ tradition. Not only did Al share an valuable programming tool, but in this and other DDJ articles he unraveled many of the twisted knots of protected- mode programming.

DOS wasn't the only operating environment DDJ covered. UNIX, embedded systems, the Mac and other platforms were an integral part of this year's lineup, along with topics like neural nets, image processing, ray tracing, and communications.

But hard code and soft tools weren't our only offerings in 1990. DDJ published one of the most year's most controversial software-related articles with the November presentation of "Software Patents," a position paper written by the League for Programming Freedom. It can

be argued that software patents are the most ruinous or beneficial developments in the programming arena—depending, of course, on which side of the patent fence you sit.

In September, contributing editors Andrew Schulman and Ray Duncan launched the "Programmer's Bookshelf," a monthly book review column. And 1990 brought into the limelight topics like windowing environments (particularly Windows 3), 80386 protected mode, and object-oriented programming, all of which were extensively covered in Dr. Dobb's.

All in all, the reference volume you hold in your hands contains all of this and more. A simple perusal of this volume's Table of Contents will fire your interest. You'll then find yourself coming back to this book many times over in the coming years.

Now we can get on with the next 15 years.

Jonathan Erickson

editor-in-chief

15 YEARS OF POWER PROGRAMMING!

#160 JANUARY 1990 $2.95 (3.95 CANADA)

NEW THIS MONTH! PROGRAMMER'S WORKBENCH

Dr. Dobb's JOURNAL

SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

REAL-TIME & EMBEDDED SYSTEMS PROGRAMMING



REAL-TIME ANIMATION

REAL-TIME DATA ACQUISITION

FORTH FOR EMBEDDED SYSTEMS

MANAGING ERROR MESSAGES

DATA ENCRYPTION

PARAMETRIC CIRCLES

STALKING GENERAL PROTECTION FAULTS

0 38351 16562 8 01

CONTENTS

JANUARY 1990
VOLUME 15, ISSUE 1

FEATURES

REAL-TIME ANIMATION 16
by Rahner James
Smooth, non-flickering, real-time EGA animation is a reality with Rahner's sprite driver. Rahner covers animation algorithm design, animation structures, and EGA innards.

REAL-TIME DATA ACQUISITION USING DMA 28
by Tom Nolan
Build your own real-time data acquisition system with the hardware and software tools Tom presents here.

ZEN FOR EMBEDDED SYSTEMS 38
by Martin Tracy
DDJ's Forth expert presents ZEN, a tiny Forth system written entirely in Forth. Programs written in ZEN are ideal for embedded applications and, says Martin, inherently ROM-able.

ERROR MESSAGE MANAGEMENT 48
by Rohan T. Douglas
Automate your error message documentation using the tools that Rohan provides here. Or, for that matter, adapt his technique for on-screen menus, prompts, and dialog boxes.

S-CODER FOR DATA ENCRYPTION 52
by Robert B. Stout
S-CODER is the core of a data encryption engine that can be implemented with virtually any high-level language and can serve as the building block for enhanced security systems.

PARAMETRIC CIRCLES 60
by Robert Zigon
Robert returns to *DDJ*, this time with an algorithm for efficiently generating circles.

LOCATION IS EVERYTHING! 124
by Mark R. Nelson
A general-purpose "locator" program that matches code and data with target hardware.

EXAMINING ROOM

EXAMINING ZORTECH C++ 2.0 64
by Scott Robert Ladd
Scott put Zortech's C++ 2.0 to the challenge of fractal geometry — and liked what he saw.

PROGRAMMER'S WORKBENCH

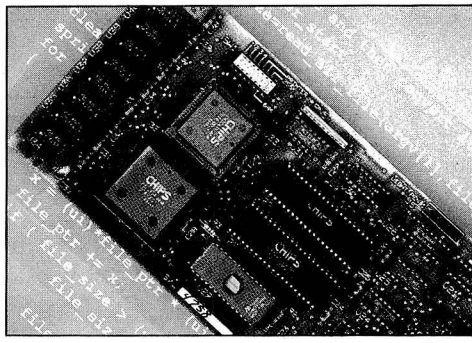
STALKING GENERAL PROTECTION FAULTS: PART I 74
by Andrew Schulman
In the first installment of this two-part article, contributing editor Andrew Schulman inaugurates a new section in *DDJ* by stalking the elusive general protection fault using a variety of developer's tools.

COLUMNS

PROGRAMMING PARADIGMS 129
by Michael Swaine
If your programming palate has been hungry for a taste of Lisp, Mike's monthly menu should fill the bill.

C PROGRAMMING 132
by Al Stevens
Al continues with his text data base indexing and retrieval project, this month adding the expression interpreter. He then reflects upon OOPSLA '89, how teachers teach C, and recommends a recent book for C programmers.

STRUCTURED PROGRAMMING 140
by Jeff Duntemann
Jeff bounces around some myths put out by object-oriented hypesters, pointing out that what you read isn't necessarily what you get, before returning to his discussion of polymorphism with Turbo Pascal and QuickPascal.



FORUM

EDITORIAL 6
by Jonathan Erickson

LETTERS 8
by you

SWAINE'S FLAMES 168
by Michael Swaine

PROGRAMMER'S SERVICES

ADVERTISER INDEX 160
where to go for more information on products

OF INTEREST 161
compiled *by Janna Custer*

PROGRAMMER'S MARKETPLACE 162
classified ads

NEXT ISSUE

In February, we'll look out on the world of windowing systems, including memory management under MS-Windows, 3-D graphics with X Windows, and windowing development tools.

Running Light, And Still Without Overbyte

Although it doesn't seem that long ago, it's 15 years ago this month that Patty Hearst went on trial for terrorist-induced bank robbery, Gerald Ford dodged golf balls while living at the White House, and Terry Bradshaw lead the Steelers to a Super Bowl victory over the Dallas Cowboys. (And if you can remember when Bradshaw had hair, not to mention when the Cowboys could make it to the Super Bowl on a regular basis, you're showing your age, just like the rest of us.)

At the same time, the plum orchards of California's Santa Clara Valley were giving way to Silicon Valley's R&D labs and the crew at an outfit called the "People's Computer Company" were putting together a "reference journal." Dennis Allison contributed the "D" from his name, Bob Albrecht handed over "ob" from his and the publication was christened *Dr. Dobb's Journal of Computer Calisthenics & Orthodontia: Running Light Without Overbyte.* Admittedly, the first *DDJ* had limited goals: To provide a minimal Basic-like programming language for writing simple programs. From the outset, the undertaking was a "participatory design project" in which readers were asked to share information and articles that made up the magazine. After the initial issue, Jim Warren came aboard as editor (for the princely salary of $350/month), and stayed around to guide *DDJ* for the next couple or so years.

Since those early days, many of the pioneers of the computing revolution have participated in the *DDJ* experience, leading us to this issue, which kicks off the 15th year of publication for the Doctor. Not only has *DDJ* survived, but it has grown, become established, and is lauded (most recently by CompuServe's *Online Today* magazine) as the "granddaddy of all programmers' magazines."

As I've said before, there aren't many magazines, computer-related or not, that manage to stick it out for 15 years. *Byte* is still there, also proudly proclaiming its 15th year of publication, and I feel particularly lucky to have been associated with both *DDJ* and *Byte.*

This past week, I called Dennis, Bob, and Jim, and all three were a little surprised to realize that the magazine was moving into its 15th year. It's not that they didn't expect *DDJ* to be around that long but, as Jim said, "we never really thought about any future for *DDJ.* Things were so chaotic back then and moving so fast that our focus was on the here and now." Bob added that "*DDJ* was supposed to be a four-issue, forty page project, and then self-destruct. But it's apparent that even back then, *DDJ* filled a need that people wanted." Dennis concurs: "*DDJ* just grew. We started it as a newsletter because there was nothing out there like it. And there still isn't. It stands almost alone among technical personal computing magazines. We xeroxed some flyers, then all of a sudden we had 400 or 500 subscribers within the first two weeks. It looked like there was an audience." Both Bob and Dennis agree that much of *DDJ*'s early success rests on Jim's shoulders. "A lot of the flavor of *DDJ* came from Jim," said Dennis, with Bob simply adding that "Jim did a hell of a job."

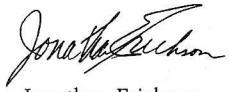
But editors move on to other things, paradigms come into and go out of vogue, and languages take on new forms. More so than anything else, *DDJ*'s one constant over the years — the real key to its success — has been the desire and openness of *DDJ* readers to share programming information and techniques with each other. Even before I joined the *DDJ* staff, Ray Duncan underscored this when he said that *DDJ*'s greatest strength was its core of faithful readers. But, as Dennis points out, that's nothing new. "From the very start," he says, "*DDJ*'s loyal following has made it special."

It comes as no surprise that Dennis, Bob, and Jim are still involved with computers and programming in one way or another. Dennis continues his lectures on computer science at Stanford, and consults the rest of the time ("You have a computer you want built, I'll build it."). Bob writes books on and teaches about programming (he's still a believer in Basic), and Jim's just completed a four-year stint as a community college trustee and is resharpening his programming skills.

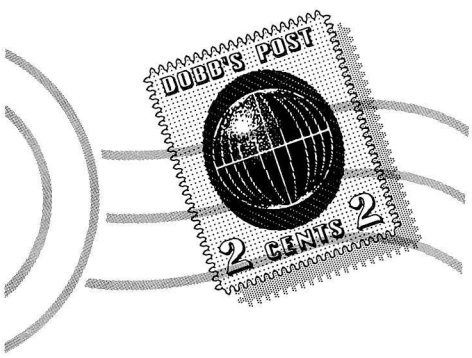
The only thing I'd like to add is a simple thanks to Dennis, Bob, and Jim, and all you readers who have stuck with *DDJ* over years.

On another subject, you'll notice that this month's issue inaugurates the "Programmer's Workbench." This new monthly section focuses on developer's tools and how they interact in a complete environment for developing a given application. The types of tools we'll cover range from prototyping tools and code generators that support the API of a complex GUI to add-in libraries that make it possible for you not to continually reinvent the wheel. Rather than simply present a shopping list of tools and a discussion of how they interact, the Programmer's Workbench orchestrates the tools on the workbench to develop something real. And in the process, the strengths and weaknesses of each tool are revealed. Ultimately, you'll walk away with a sense that you actually used those tools, and you'll have the code to experiment further.

Contributing editor Andrew Schulman leads off with a two-part article, exploring the problem of general protection faults. If you have ideas for similar Workbench topics and tools, we'd like to hear from you and spread your tools out across the workbench as well.



Jonathan Erickson
editor-in-chief



Stymied by C

Dear *DDJ*,

I thoroughly enjoy Jeff Duntemann's "Structured Programming" column in *DDJ*. It's always easy to enjoy someone I agree with! Besides having a fondness for structured programming and Turbo Pascal, I also share his coolness towards C. Which brings me to the main reason for this letter. I have a software project where I must use some C routines provided for me in .LIB files. I have Microsoft QuickC out of necessity, and Turbo Assembler/Turbo Pascal 5.5 out of preference. What I would like to do is somehow hook the canned C routines from Turbo Pascal. The kind of applications I'll be working on lend themselves well to objects, as does my way of thinking. My hope is that there is some way to build a TP unit that has TP procedures/functions that actually are calling the C functions. Once this unit is established I could happily turn my back once again on C. What I'd have is a reusable unit with the C hidden an arm's length away.

Is a hook like this possible? Perhaps some assembly interface calling the C and in turn being called by the TP code? I don't have access to the C library source code itself but maybe I could write enough C code to call the canned functions with built-in hooks? I've done more in TP than either C or assembly but have dabbled in both. I have worked with externals written in assembly and called from Turbo Pascal, modifying already written code, but need a little hand holding if this is the route to go with this problem. I'd really appreciate any help Jeff could offer.

Dale Lucas
Cedar Rapids, Iowa

Jeff responds: Without being able to modify and recompile your C libraries, you're stuck. It's a matter of who cleans up whose messes. When C code calls a C function, the caller sets up the stack frame, calls the function, and then (when the function returns control to the caller) the caller cleans up the stack by removing parameters.

Pascal, on the other hand, requires that the function (or procedure) clean up the stack before returning control to the caller. The caller assumes that it receives a clean stack after each call. The two systems are completely incompatible, although, some C compilers, including Microsoft's, allow you to specify Pascal calling conventions as an option. If you could recompile your C code, you could invoke this option and make the two worlds coexist peaceably.

C does what it does to allow the number of parameters passed to a function to vary from one call to the next. To each his own; some people eat fugu and deserve whatever they get. I take some comfort in noting that operating system API calls use the Pascal calling conventions; evidently OS architects draw the line at risking the system's neck for this kind of silliness.

But that doesn't solve your problem. If you only have .LIBs, well, shucks, you have two choices:

- *Become a sadomasochist, embrace the C language, and work twice as hard as you need to in order to get your work done for the rest of your career. On the plus side, this entitles you to wear a T-shirt reading, "Look what a big tough macho hacker I am!"*
- *Give your gracious benefactor his C code back and recode the routines from the interface spec in Pascal. Dimes to donuts you'll spend less time recoding the C routines in Pascal than you would recoding your app in C.*

Maybe I'm only half-a-hacker . . . I like to take a little time now and then to neck with my wife, build radios, and throw the rag for Mr. Byte. I get my C in orange juice. I'd advise you to do the same.

DDJ Passes the Acid Test

Dear *DDJ*,

Many times during the last year I was on the verge of deciding to let my subscription to *DDJ* lapse, as it had apparently turned into a fanzine for 386 and DOS groupies. One more article on cute tricks in C, how to write TSRs, and the 640K barrier would have done it.

Then Michael Swaine's interview with Hal Hardenbergh made me give it one more month. The September issue made me give it another year, and I have sent in my renewal. Five out of six articles were of interest to non-DOS, non-PC programmers!

For 1990, please make at least half the articles in any issue independent of platform and language, i.e., of interest and value to programmers on any machine using any language.

If you wonder where I am coming from, I am a chemist; I write about 10,000 lines of code a year for my research, in Pascal (by preference), C (by necessity), and (ugh!) Fortran. I use a VAX station and a networked VAX cluster for programming, PCs for document processing, and Macintoshes for making slides and figures. I am currently using neural nets and simulated annealing. I am sure you have many readers who, like me, do no programming on PCs, but who are always on the lookout for new programming techniques and useful algorithms.

Ernest W. Robb
Glen Rock, New Jersey

DDJ responds: You've hit on what DDJ *is really all about, Ernest — sharing new programming techniques and handy algorithms — and we'd like to hear from anyone who has some to share.*

DFS At Work

Dear *DDJ*,

I read Rabindra Kar's article "Data-Flow Multitasking" in the November 1989 issue of *Dr. Dobb's Journal* and enjoyed it very much. The title of the article attracted my attention first because, as part of my job for the past three years, I designed and was principal developer of a data analysis system based on what he termed data-flow multitasking (I've been calling it data-flow processing). Currently this system consists of 58 fittings (programs which tap into a data flow) with support libraries consisting of 264 functions. Fittings may be written in either C or Fortran, and the system runs under Unix and VMS.

This system (which I call the Data-Flow System or DFS) takes a slightly different approach to establishing the aggregate task. Instead of making each processing element of the task a function in an application, each element is itself an application which can be "connected" to any other element. The main reason for this approach was to make it easier for the user to construct aggregate tasks without the need to deal with a programming language. This approach also allows fittings to be written in the programming language which best implements the task.

The variety of fittings in this system is quite broad. There are fittings for reading and writing data, analysis, data-

(continued on page 12)

(continued from page 8)
flow management, and data visualization. It is also easy for a user to develop a new fitting to match any special needs. The DFS also has several different interfaces; they include a batch type interface, an interpreter, and a fully menu-driven interface. We have also developed a specialized interface which a user can navigate through and create an SQL query. This query is then integrated into an aggregate task, which extracts the data from one or more data nodes in a network (a distributed data base) and then presents the data locally for viewing.

The system I've described is in use where I work (The Institute of Geophysics and Planetary Physics at UCLA) and at several other locations which are involved with space physics research. I'd be happy to supply more information if anyone's interested. I must compliment Robin on the clarity of his presentation and the succinct way in which he described the concept. It was a very good article.

Todd King
Los Angeles, California

C Dynamic Strings

Dear *DDJ*,
Just finished Al Stevens's column in the October issue, with special interest in his item on dynamic strings in C. The ease of string manipulation in Basic is one of the reasons I keep using it and haven't switched entirely to C (besides the fact that I still have to keep the old Z80 Radio Shack alive for my wife).

A few years ago I picked up Alcor C for the R/S. Never did anything big with it because I couldn't fit much into a 30K workspace anyway . . . . When I saw Al's item on strings I dug up the old manual because I remembered that Alcor had dynamic strings in those old days using a structure defined in *stdio*, along with a couple of conversion functions and extensions of *printf* and *scanf*. Their implementation was a simple structure with a length byte and a char array, but it worked.

Alcor C is still around. I think it's MIX C now; at least the company is in the same town and still lowballs everybody on price. It was available back then for Apple II, CP/M, and Radio Shack machines. I got it originally as a learning tool, which it served well as. It was not industrial strength; any word on the condition if MIX these days? At the price ($19.95 last I saw) it must be worth something.

Al, keep up the good writing. By the way, this comes via US Mule because I couldn't find a CompuServe address for you or *DDJ*. I did find the *DDJ* Forum, but no listing for Al to send a message.

Michael Brady
Fresno, California

Thanks, Michael. Al's CIS address is 71101,1262; DDJ's *is 76704,50.*

And NewWave Begat. . . .

Dear *DDJ*,
In regard to Michael Swaine's column titled "Unbundled Integration," in the August 1989 issue of *Dr. Dobb's Journal*, I would like to make the following observations. The "most interesting" feature of the Apple System Version 7.0 release, the InterApplication Communications architecture (IAC) is indeed an interesting development. Over a year ago, Hewlett-Packard demonstrated this capability as a feature of its NewWave environment. (I guess imitation really is the sincerest form of flattery, especially when the imitator has pending legal action against the imitatee for imitating the imitator!) NewWave is part of the user interface proposed by the Open Software Foundation as Motif.

HP's term for dynamic cut-and-paste is "shared links." With NewWave, a user "shares" and "establishes a view to" data, while an IAC user "publishes" and "subscribes to" data. It probably makes little difference to a user which set of new terms must be learned to use this feature, but I think I'd prefer almost any set of terms that doesn't remind me of the recent spate of renewal notices I seem to keep getting from computer industry publications, in direct proportion to the number of years for which I've renewed my subscriptions to them! The important thing to remember is that it looks like the folks responsible for these windowing environments are making good on yet another implied promise — software that remains integrated even after one program changes. More power to them all and the Flames of Swaine!

Lawrence T. Prevatte, III
Cape Canaveral, Florida

Finite State's Rights

Dear *DDJ*,
Donald Smith's article (October 1989), "Finite State Machines for XModem," was a timely help. I recently proposed an FSM design to a client who had not heard of FSM. Donald's article added credibility to my proposal.

I have used FSMs to manage communication protocols, to manage user interfaces, and to implement control logic. Using FSMs often, I implemented a translator and C library called "The State Machine," which is just what Donald suggested as a coding technique. Interestingly, the code that this translator generates looks remarkably similar to Donald's hand-coded state machine. I have developed an example, "MENUS," which uses an FSM to control the navigation about a menu-driven user interface. The state diagram that was the original design document for this example was directly encoded into a language called "State Transition Description Language" (STDL). The state machine has a number of unique features: It aids memory reduction and improved productivity by supporting encapsulation with both macros and callable substates. The capability to call substates is very useful when implementing user interfaces with submenus. Debugging is simplified by a trace option which can generate a log of all state transitions that occur during execution. This option can be turned off to minimize memory requirements. Actions are C functions which can have any number of arguments. An event or stimulus is defined by a Boolean C function. The state machine comes with several predefined action and event functions.

There are many parallels between the implementation philosophy of the state machine and the mini-interpreters described by Abrash and Illowsky in their article (September 1989), "Roll Your Own Minilanguages with Mini-Interpreters."

For those with a champagne budget, there is a sophisticated CASE tool available from i-Logix called "Statemate." It has several powerful features: Statemate is a graphic tool. State diagrams are drawn interactively. The tool integrates a data base and an interpreter that supports complex events and actions. Documentation can be generated automatically from the data base. State diagrams can be executed using Statemate's simulator or they can be executed from C or Ada code generated by Statemate's prototyper. FSMs defined by Statemate can be decomposed. This capability is based upon a rigorous mathematical theory developed by David Harel. It is important for complex systems which would otherwise explode into hundreds of states.

Rob Buck
Aerosoft
Fairfield, Iowa

Graphics For the Rest of Us

Dear *DDJ*,
I found the interview with David Parker in Michael Swaine's column entitled "Parker's Perceptions" in the October, 1989 *DDJ* especially interesting since I'm a very satisfied user of his "Ac-

(continued on page 14)

(continued from page 12)
roSpin" graphics software. I'd known nothing about the person behind it!

Swaine says "it [Acrobits] seems to have a good product," and I can testify that it does indeed. My colleagues and I need to do 3-D graphics in a variety of languages, on a number of different IBM PC compatible machines. We also cannot afford fancy hardware; my own machine doesn't even have a hard drive.

AcroSpin is the only product I've seen that does the things I need on the hardware I've got. On all our PCs at this institution, I've yet to encounter one that has some kind of graphics and cannot run AcroSpin. It's clean, simple, and does exactly what the manual says it'll do.

Matthew D. Healy
Zoology graduate student
Duke University, North Carolina

Down Those Hallowed Halls

Dear *DDJ*,

Funny how time alters everything. When Jeff Duntemann opened his September 1989 "Structured Programming" column with a mention of his high school days, I had to pull out my copy of the Lane Arrowhead '70 and look him up. If he still has his copy, he can find me on pages 27 and 126. He must have picked up on computers in college.

OOPs, it still leaves me in the dark. I guess for my PUNishment, I will have to rewrite Fortran into Basic. I enjoyed his column. Take care.

Thomas Kocourek
Carrollton, Georgia

Magnitude, Made to Order

Dear *DDJ*,

In Michael Swaine's November editorial on innumeracy, his accusation of an error in John A. Paulos's book, *Innumeracy*, is based on a misunderstanding of the term "order of magnitude." He starts off okay, by stating his deduction that Paulos uses the expression "order of magnitude" to mean "a power of ten." He goofs when he asserts that magnitude (plural) means "powers of one hundred." This is wrong. Any scientist will tell you that scientists use the term "orders of magnitude" to mean "powers of ten," for example:

If quantity A is ten times greater than quantity B, then A is said to be one order of magnitude greater than B. If A is one hundred times greater than B, then A is said to be two orders of magnitude greater than B.

Getting back to the example of the malaria safety index that Swaine cited, for the malaria index to be "orders" of magnitude lower in most of the world than in the United States merely means that the actual risk of malaria (defined as the total population count divided by the number of people in that population who catch malaria) is at least one hundred times higher (i.e. at least two orders of magnitude higher) than in the United States. (Remember that on Paulos's scale, the lower the index number, the greater the risk.) By using the correct definition of "orders of magnitude" we see that Paulos's assertion is indeed within the realm of plausibility.

Swaine's misunderstanding of the term "order of magnitude" is understandable, because written definitions of this term in dictionaries (even scientific ones) are hard to come by. The use of orders of magnitude in calculations has been spread by word-of-mouth among scientists and has not been exposed to the general public (except by Carl Sagan in his *Cosmos* series). Despite the mistake, I thank Mr. Swaine for addressing the issue of innumeracy in his editorials.

Gregory B. Goslen
Research Triangle Park, N. Carolina

Zortech Heard From . . .

Dear *DDJ*,

We read with interest Al Stevens's column on C++ in the October issue of *DDJ*. Imagine our surprise, when on pg. 128 we read the statement: " . . . not until Borland and Microsoft introduce C++ compilers, complete with integrated development environments and hot-shot debuggers, can PC developers get serious about it."

Are you actually saying that only a couple of vendors are allowed to produce a C++ system for MS-DOS? That sort of statement belongs in company marketing literature, not in a well-respected publication like *DDJ*, where developers look for impartial editorial content.

Zortech has over 50,000 users now, including virtually every major corporation. Many are switching to the Zortech C++ compiler as their preferred development tool. How much more serious can developers get than to switch from C to C++ for their next generation products? This is happening now, and we would be delighted to provide Al with a list of major corporations who are developing with Zortech C++.

Incidently, the "integrated environment and hot-shot debugger" requested by Al is now available with the release of Zortech C++ V2.0 Developer's Edition. It is a full-blown development system including the world's first MS-DOS C++ Source-Level Debugger, AT&T C++ V2.0 compatibility, and many other enhancements. Our entire company is focused on C++. It isn't a sideline for us and doesn't play second fiddle to other languages, wordprocessors, spreadsheets, or database packages.

Walter Bright
Director of Technologies
Zortech Inc.

The Foot Bone's Connected to the Head Bone

Dear *DDJ*,

The correct answer to the question posed to Michael Swaine in New Orleans (December 1989 *DDJ*), "Where you got your shoes?" is "You got your shoes on your feet!"

I had to tell you . . .
Harold O. Koenig
Scottsville, Virginia

Disrobing the Emporer

Dear *DDJ*,

Thank you for publishing Mr. Guthery's informative article, "Are the Emporer's New Clothes Object-Oriented?" in the December 1989 issue. Amidst all of the hype, it's good to see some reason appear. Now, I can say: "OOPS! Did they really do that?" Rather than being an advance, object-oriented programming seems to be regressive.

Regressive? Yes, it's a return to "spaghetti code" and violates the basic tenets of writing well-structured programs. It was this type of violation by the excessive use of GOTOs that led to Dykstra's letter to the ACM in 1969 and the ensuing interest in structured programming.

In well-structured programs, the structure is a tree whose nodes are the program modules. Each node in a tree may have one and only one father, but, of course, may have has many children as necessary to do the job. As defined in the literature, an object is a module that may receive a message (be called) from any number of sources and may send messages to (call) other objects. Thus, an object is a structural node that may have more than one father — a rather unnatural situation! Can you imagine the difficulties when trying to debug a module entered from who-knows-where?

Instead of coming up with new buzz words like OOP, the language of developers should heed our desires for more granularity in the libraries associated with various compilers. This was pointed out admirably by Bruce W. Tonkin in his "Examining Room" column on PDQ in the same issue.

If you keep coming up with articles like the two I've mentioned, I'm afraid I'll have to subscribe to *DDJ*. Thanks.

Dan W. Crockett
Queen Valley, Arizona

Real-Time Animation

Presenting a sprite driver for EGA

Rahner James

"The distinction between a toy and a game is that the game has a goal; therefore, life is not a game, it is a toy." — GOK

As a child, I could not differentiate between Bugs Bunny and Walter Cronkite. This is not to say that the man America most trusted had dental problems, but that the child did not have the experience to see the cartoon for what it was — a stream of individually drawn pictures. Skilled professional help was required to deal with the trauma caused by the revealed truth. Even after the psychological defects were converted to scars, I had to wait until I could create two-dimensional life for myself.

The Time Has Come

Animation is achieved by showing a series of incrementally changing images. Depending on the duration of time that a single image is shown and the time it takes to switch to the next image increment, the viewer's visual persistence smooths out the image's transition. Anyone with a compiler and a graphics library can put a series of images on a computer display. But in order to create smooth, non-flickering, real-time animation, a fair amount of thought is necessary.

To support reasonable animation, the algorithm must conform to the following rules:

Coordinate Movement — The routines must allow individual objects to be moved around on the screen and placed on any pixel boundary.

Independent Motion — Each object in the image should be able to show a chain of sequences independent of its coordinate movement.

Smooth Transitions — The transition between image frames should have no intermediate stages. This means that the viewer should see only complete images, not images that are half the first image and half the second image. These half-and-half images are perceived as a flicker and are distracting.

Regular Transitions — All image transitions should occur at regular intervals. If this does not happen, the sequence will appear to jerk as if shown on an old projector.

Sprites — If an object has a hole, any objects that are behind it should show through. Poorly done animation will not allow the viewer to see through a gap in an object.

Realistic Objects — There is a difference between what is shown on the screen and what is perceived by the observer. Up to a point, jagged lines will be smoothed, imperfect colors accepted, and stairstep corners rounded. This point, the point of realism, is subjective and entirely dependent on the target audience. Below this point, other unrelated flaws can be magnified; above this point, the overall perception of the product is enhanced.

The elements of this list are all mutable by targeted hardware limitations. Accurate shading may be difficult on an AT-class machine. Absolute realism for anything but the simplest geometric shapes is impossible on CGA or EGA. The finished product's design constraints may adjust the significance of one element over the others. These are Rahner rules — they may be burnt, bent, or beatified.

Zippy Tries His Hand at CGA Animation

When I first saw reasonable animated images on a CGA adapter, I was intrigued. It was a simple CGA sprite demo —

Rahner is an independent consultant living near Sacramento, Calif. He can be reached by phone at 916-722-1939 or through CompuServe at 71450,757.

several helicopters flying aimlessly around the display. Because the demo allowed the helicopters to start only on a byte boundary (four pixels per byte) and did not allow them to exit smoothly from the side of the screen, I decided to try my hand at writing the ultimate animation driver for CGA.

After a lazy Sunday's work, the driver was finished. Given CGA's limitations (four colors and 320 × 200 resolution), ultimate is probably too strong a word. It was about the same as tying the ultimate shoelace or throwing the ultimate dirt clod. Anticlimactic would be the correct word for polite company.

Because the initial attempt was an unplanned, seat-of-the-pants, I've-got-Fritos-diet-Coke-and-plenty-of-time effort, my trial and error path would best illustrate some animation rudiments.

The first attempt was to whip a series of changing pictures past the monitor. This involved copying virtual screens, which I had prearranged, onto the CGA video RAM. This had the interesting effect of flickering the screen with seemingly random images caught in a blizzard.

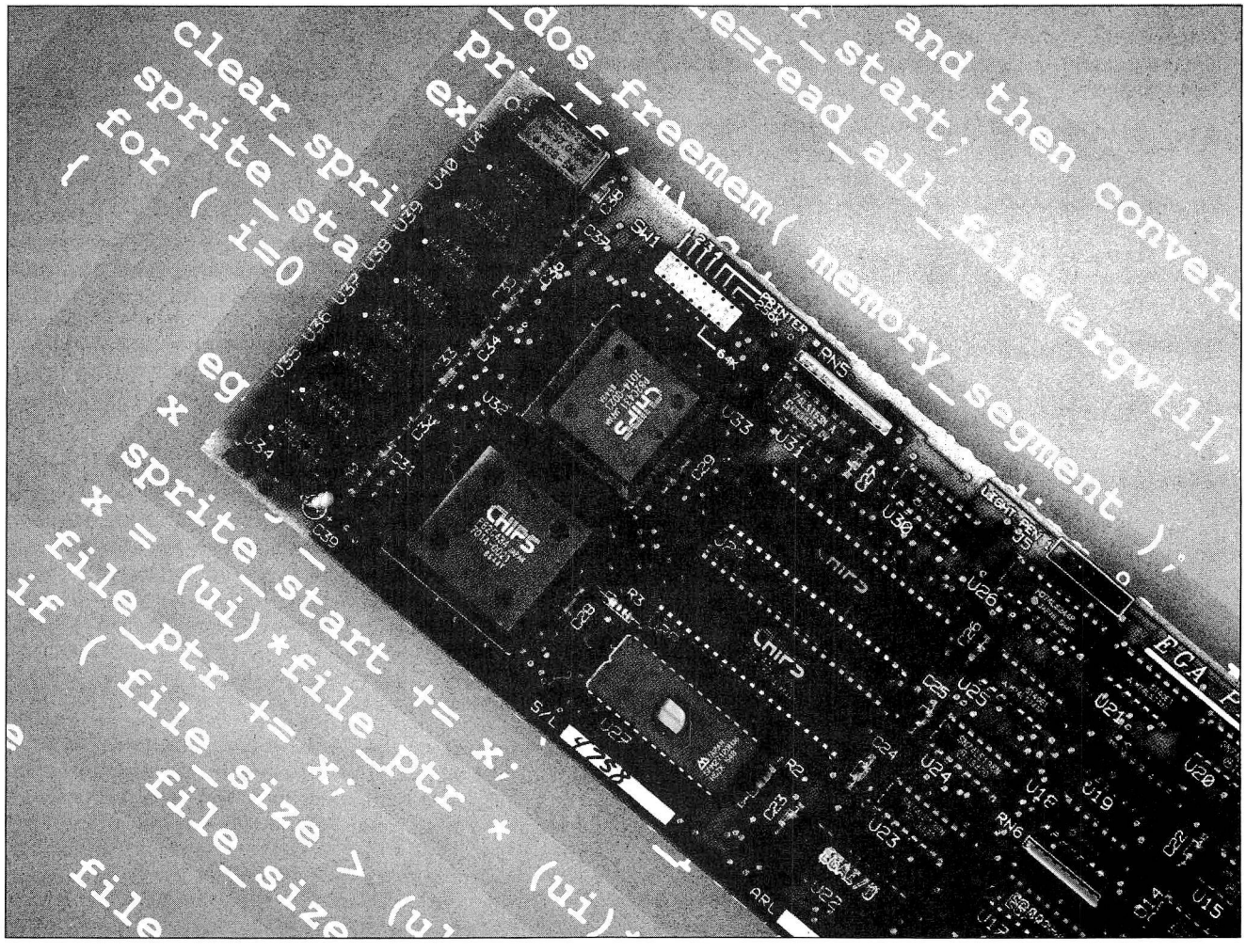
The flicker and snowstorm were due to not waiting for the vertical retrace before displaying the next image. What is vertical retrace? A good question, because it is important later. The CRT monitor etches its pictures in the orbitals of fluorescent compounds with a single beam of electrons. This beam sweeps horizontally back and forth across the screen. In either direction — back or forth — the beam has to turn off, otherwise it looks funny. The off direction is called the horizontal retrace. The beam winds its way down the screen in this back and forth manner. When it reaches the bottom, it turns off and returns to the top of the screen: This off time is called the "vertical retrace." On the standard EGA setup, this vertical retrace occurs 60 times a second.

The standard CGA adapter has a bit that indicates when the horizontal refresh is occurring and another to show when the vertical refresh is occurring. It was a simple task to change the program so that it waited for the vertical retrace before blasting out the next virtual screen. — I saw a series of pictures endlessly circling, but without snow or flicker.

Next I noticed that most of the screen was stationary and only a small fraction of the image moved at any one time. In fact, most of the things that did move were sets of pixels that did not reposition themselves with respect to one another, only with respect to the rest of the picture. After much deliberation (and lunch), I named these sets "sprites." Later I found that these sets had been noticed by others before and they had used my name for them. Rather than risking a protracted legal battle, I swallowed my pride and have allowed the others to take the credit.

The basic concept behind the sprite is simple. Cut a rectangular section out of the screen and store it. Then take the set of pixels that comprise the sprite and replace the cutout section with them.

Armed with my "new" creation, I reduced the series of virtual screens to a simple background and a few sprites. The background was displayed first, then the sprites were moved into position before the finish of the vertical retrace. Now I had the same cinematograph that I had before, but the program was more efficient and the storage require-



ments were reduced. Before I got bored with this endless video cycle, I noticed that when my sprites went in front of something, the background was completely covered, even in places that I should have been able to see through.

This meant another change. Instead of just storing the background, I masked off the solid areas of the sprite body. Instead of replacing the cutout, I ORed the sprite onto the masked background, then replaced the cutout around the finished product. This allowed me to have "holes" in the sprites, for increased realism.

Absolute realism for anything but the simplest geometric shapes is impossible on CGA or EGA. The finished product's design constraints may adjust the significance of one element over the others. These are Rahner rules–they may be burnt, bent, or beatified

With the inclusion of holes in the sprites, I had my ultimate CGA sprite routine. All the little fishes were swimming around in my video aquarium without the need for food. I was satisfied and went to bed.

You may be asking, "What does this have to do with EGA?" You may be getting sleepy and ready to close the magazine. You may just be hunting for good prices on software. Well, to the shopping sportsman, there are no good prices in this article; to the somnolent peruser, good night; and to the inquisitor in the back with his hand up, Everything!

The EGA and I

In its 640 × 350-pixel color graphics mode, an EGA adapter with 256K of RAM is set up as four planes of 28,000 bytes each. It also has two pages, one that is being viewed on the monitor and one that is in the ether. Both pages can be addressed directly by the CPU. The first page starts at memory address *A000:0000h* and the second page starts at *A000:8000h*. Each byte of EGA memory represents eight pixels with the most significant bit (MSB) being shown as the leftmost pixel. A byte or bit of any combination of the planes may be addressed depending on how the EGA registers have been set.

Superficially, there seemed to be little difference between a sprite driver for the CGA and EGA adapters. It seemed to be just another block of RAM that I needed to jam out bytes. Following that line of thought, the first code translation from CGA was conceptually simple. The sprites were placed on the visual page, a plane at a time. When I looked at the result for the first time, I found myself almost back to square one. No matter how efficiently I wrote the driver, there was a constant flicker. I ran to the bookstore, hoping for a tome of enlightenment. My hope was dashed by a limited selection. But a quick reread of the *IBM EGA Technical Reference* manual provided me with the answers: The EGA adapter can generate an interrupt and the visual page can be switched during the vertical retrace.

Writing a Bit Map to EGA Memory

The EGA adapter has a fair number of features. All the features and the way the board reacts to CPU memory manipulations are determined by the configuration registers. Most of the registers are set up in pairs. The first register accepts an index value that determines the functionality of the second register. The major register pairs that we need to concern ourselves with are the Sequencer registers and the Graphics 1 & 2 Address registers. Another register that is important for this discussion is Input Status Register One.

The Sequencer register is located at *3C4h* with its index register at *3C5h*. It has five indexed registers: Reset (0), Clocking Mode (1), Map Mask (2), Character Map Select (3), and Memory Mode (4). To access an indexed register, the index's number is output to the Sequencer register followed by the value for that index output to the index port. For example, if you want to place a 5 in the Clocking Mode index register, which is index number 1, the assembly code in Example 1 would do. Because all the register pairs are one right after the other, this code segment could be replaced by that in Example 2. This replacement is usually valid, except when slow ports cause timing difficulties.

The important Sequencer index register, with respect to our driver, is the Map Mask Register (index 2). This register enables planes so that the CPU can write to them. Setting bit 0 enables plane 0, bit 1 enables plane 1, and so on. Because there are only four planes, the four MSBs are not used and are ignored. If you wanted to write the same information to multiple planes, multiple bits could be set. No easy assumptions can be made about the sprite data, so we can't really take advantage of this feature.

The Graphics 1 & 2 register set deals with colors, pixel masks, and the Boolean graphic operations the EGA can perform. This register is configured the same as the Sequencer register, with nine indexed registers: Set/Reset (0), Enable Set/Reset (1), Color Compare (2), Data Rotate (3), Read Map Select (4), Mode Register (5), Miscellaneous (6), Color Don't Care (7), and Bit Mask (8). The index registers of concern are the Data Rotate register and the Read Map Select register.

The Data Rotate register has two controls. Bits 0 – 2 represent the Rotate Count. The Rotate Count is a binary encoded number that represents the bit positions to shift any data written to a video plane. Because all our data will be unshifted at the hardware level, this value should be 0. Bits 3 – 4 represent the Function Select. The Function Select indicates which Boolean-type operation is desired for pixels

```
MOV     DX, 3C4h        ; DX -> Sequencer register
MOV     AL, 1           ; AL = index 1, Clocking Mode
OUT     DX, AL
INC     DX              ; DX -> Sequencer index port
MOV     AL, 5           ; AL = to put in Clocking Mode
OUT     DX, AL
```

Example 1: *Sample code for writing bit map to EGA memory*

```
MOV     DX, 3C4h        ; DX -> Sequencer reg. pair
MOV     AX, 501h        ; AL = index 1, AH = value 5
OUT     DX, AX          ; Puts AL out 3C4h, then
                        ;  AH out 3C5h
```

Example 2: *Replacing the code in Example 1*

written to display memory. Table 1 shows the available functions.

To diverge for a moment, a definition for "latched data" is in order. No matter how it appears, the video memory on the EGA adapter is never directly connected to the PC bus. When the registers are set properly, the program addresses the video memory in exactly the same manner it would any other portion of main memory. The memory can be accessed as a byte or a word, but those accesses are processed through the EGA's circuitry. The circuitry performs some gyrations on the data, then passes it on. In order to properly swing the binary song, that gyrating EGA circuitry latches the byte or word in its internal read/write buffer. A read of EGA memory will put that byte of pixels in the latch, which can then be operated on by some future operation. Because each plane has a separate latch buffer, if all four planes have been enabled, 32 bits at a time can be latched (read), operated on, and then rewritten to EGA memory with a single 8086 instruction.

Give me an inch and I'll take a while, diverging on to the 8086 instructions. When dealing with this aspect of the EGA adapter, a close look at how some 8086 instructions actually work would be in order. Let's start with the instruction:

```
OR    [DI], AL
```

When the 8086 sees this instruction, it loads the value pointed at by the register *DI* into the 8086 internal register, ORs the value in register *AL* onto that internal register, then writes the result back out to the location pointed to by *DI*. If *DI* happened to point to EGA memory, this would latch up a number of pixels, add in pixels, then write the latch data back out to video memory —all in a single instruction. If an additional EGA function, such as a bit rotate, were added to the previous example, some interesting and possibly useful results could be achieved. I don't use this in my routine, but, by jingo, it's just too nifty to be ignored!

The last register of importance is Input Status Register One. It has a few informative bits, but we will be concerned with only the Vertical Retrace bit (3). As the name implies, this bit is set to 1 when the display is in a vertical retrace time. As I stated before, this was the time to write to the video memory with the CGA adapter. It has approximately the same value with the EGA adapter, but not exactly.

When Blazing Fast Is Not Fast Enough to Start a Fire

When I converted my CGA animation routines to work with the EGA, there was an unexpected problem. I found that no matter how fast I blasted my sprites out to video memory, the raster line (another name for that beam of electrons described earlier) would catch up to where I was writing pixels. When the raster caught up, the screen would flicker annoyingly. Having to deal with four planes and EGA's higher resolution just took too much time. I made my code the most efficient assembly routines I could. I made assumptions about the data I was displaying in order to cut corners. I got a 25-MHz 386 system. Nothing worked. I felt like a laundry soap commercial. I went back to the *EGA Technical Reference* manual.

Value	Description
0 0	Written data is not modified
0 1	Written data is ANDed with latched data
1 0	Written data is ORed with latched data
1 1	Written data is XORed with latched data

Table 1: *Available functions*

Almost immediately the answer, written by the IBM ancients, made me question what I had been thinking about in the first place. The standard EGA has 256K of RAM — enough for two pages of display memory. I could write to one page, wait for the next vertical retrace, then swap pages. I rewrote everything.

Planning ahead, I continued reading the *Technical Reference* manual. The EGA can generate an interrupt request 2. If the driver could just swap pages whenever the video went into vertical retrace, then I wouldn't have to waste time polling. I rewrote it, again.

To become animated objects, sprites must have four basic degrees of freedom: Coordinate Motion, Self-relative Motion, Rotation, and Perceived Distance

Animation Structures

To become animated objects, sprites must have four basic degrees of freedom: Coordinate Motion, Self-relative Motion, Rotation, and Perceived Distance. Coordinate Motion is simply the movement from one point on the screen to another. Self-relative Motion is the movement that the sprite could make without moving to a new coordinate location. Rotation is rotation of the sprite around some center point in its body. Perceived Distance is basically sizing the sprite according to its apparent distance from the viewer.

The increment resolution of each degree of freedom is independent of the others. A sprite picture of a person may be pumping its arm up and down a pixel at a time and traversing the screen five pixels at time. Given a monitor/graphics adapter combination that refreshed the screen an infinite number of times, the smaller the movement increment, the more realistic its action would be. Because the standard EGA board refreshes the screen at 60 Hz (60 times a second), the movement increment should be judged relative to the apparent velocity of the sprite, the display resolution, and the level of the art. Because I can draw only crude stick figures, my resolution granularity can be boulder-size.

Two of the four degrees of freedom (Rotation and Perceived Distance) should not be a function of a sprite driver. A good, general-purpose rotation algorithm requires fairly heavy calculations. These calculations are burdensome enough to detract from the real-time nature of the animation driver. Although Perceived Distance does not need as much time from the CPU, it should be done at a higher level than the driver. Perceived Distance requires the sprite to be resized larger as it gets closer to the viewer and smaller as it gets farther away. As the size of the sprite approaches the minimum resolution of the monitor, details disappear. No algorithm can make perfect decisions about which features of an object are important to the visual integrity of that object.

Self-relative Motion deals with the movement of each of the individual pixels of the sprite with respect to each other, but not straying outside the boundary of the sprite. To illustrate Self-relative Motion without any other component, I have included a sprite of a flame. Each of the pixel groups that represent small flamelets rises to the top of the fire. The

(continued from page 20)
pixel groups that represent the edges of the flame billow in the updraft caused by the heated air. In my routine, the effect of the motion is created by a linked list of sprite frames. In the example, each successive frame shows the flame in the next point of time (without regard to mathematical proofs, in animation there is a quantum of time). Because motion in most biological or mechanical systems is cyclical, I join the terminal points of this linked list into a sprite circle. Each sprite can proceed through a cycle of self-relative motions whose complexity is determined by the circumference of the sprite circle.

Coordinate Motion involves moving the sprite circle from one point to another on the screen at some regular velocity. The sprite velocity is determined by the number of pixels that the sprite circle will move divided by the number of times per second the visual image will be changed. Say we make a sprite representation of a five-meter-long car that is drawn 32 pixels in length. To move that car from the right side of the screen to the center at an apparent velocity of 20 km/hour, the sprite would have to move to the left 176 pixels per second if everything is kept to scale. If our visual page changes come at 20 per second, the sprite would have to be moved nine pixels to the left for every page change.

How the Routines Work

The sprite routines are broken down into two parts. The portion that deals with the EGA ports, memory and interrupt service is written in 8086 assembly language. Listing One (page 82) shows the EGA sprite drivers and Listing Two (page 88), the sprite circle handler. The higher-level sprite circle and list managers are written in Microsoft C. Listing Three (page 92), SPRITES.C, displays a sprite file on an EGA screen, and Listing Four (page 93) is the make file.

Some assumptions were made about the nature of the sprites and the background. The sprite driver is written for sprites of any dimension, but the driver is optimized for a sprite that is 32 bits across. In my application, it was assumed that the observer could pan or tilt the viewing perspective. Because the perspective can be changing in smooth real time, the background is not a set quantity — it is being regenerated in every visual frame. Additionally, because the 8086 family drops at least eight clock ticks every time it makes a *JMP* or *CALL*, the code favors execution speed (that is, very few jumps, calls, or loops) over program size.

The basic algorithm is simple. Before any operations can be performed, the sprite driver needs to be installed using the function *EGA_INSTALL( )*. This preps the adapter, initializes some variables, and installs the interrupt vector. The first sprite of a sprite circle is inserted into the linked list of circles by calling the function *INSERT_SPRITE (START_X, START_Y, END_X, END_Y, SPEED_X, SPEED_Y, DEPTH, SPRITE_ CIRCLE)*; where *START_X* and *START_Y* are the starting *X,Y* coordinates of the sprite, *END_X* and *END_Y* are the ending *X,Y* coordinates of the sprite, *SPEED_X* and *SPEED_Y* are the amounts that the *X* and *Y* coordinates will change per visual frame, *DEPTH* is the perceived distance from the observer, and *SPRITE_CIRCLE* is a pointer to the first entry of the sprite circle. Additional sprites can be added to the circumference of a sprite circle with the function *ADD_SPRITE*. Once all the sprite circles have been figured out and inserted into the sprite list, call *DO_SPRITE_ LIST( )* whenever it is appropriate. *DO_SPRITE_ LIST( )* figures out the new sprite positions and places them on the nonvisual page. When it has placed all the sprites, it sets the flag *DO_PAGE_ FLIP*, clears *DONE_PAGE_ FLIP*, and returns.

The main program body is then free to do anything with the sprite structures. At some time in the future, the EGA's vertical interrupts and the interrupt service routine decides whether to swap pages or not. If it does swap the pages, it clears *DO_PAGE_FLIP* and sets *DONE_PAGE_ FLIP*. Although the routines are not completely reentrant, they are fairly immune to interrupts; they can be called from other interrupt service routines such as the timer tick or a mouse driver.

Animation is achieved by showing a series of incrementally changing images

With regard to visual timing, movies project 24 frames per second on the silver screen. To decrease the cost of animation, some cartoon manufacturers will keep the same picture on the screen for more than one frame. The interrupt service routine has a counter that can be used to allow it to skip any number of vertical retraces between page changes. If your CPU is slow or the main body of your program needs more time to do its work, altering the skip count has the effect of smoothing out the movement of the sprites. To counteract the slowing effect that increasing the skip count would have, you must increase the velocity of any coordinate motion proportionately.

Finishing Thoughts

So much can be written about real-time animation that a conclusion at any point leaves us feeling a lot was left out. The scope of this article does not allow me to explore all the avenues with the depth they deserve. Maybe you can use the routines that have been provided with this article as learning aids to go beyond what has been written. Animation is a form for the presentation of ideas. Seminars, product demonstrations, and computer modeling programs can all be enhanced by the addition of animation graphics. Of course, the most obvious use only enhances what I have always said: The only useful thing someone can do with a computer is play a game on it.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 82.)

Vote for your favorite feature/article.
Circle Reader Service **No. 1.**

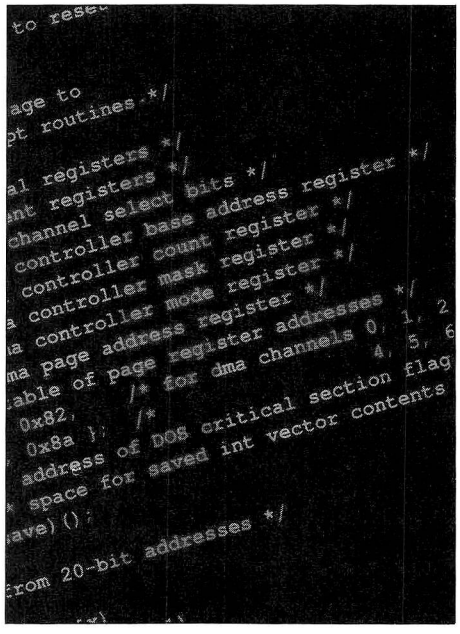
Real-Time Data Acquisition *Using DMA*

Here are all the tools — hardware and software — you need for your own data acquisition system

Tom Nolan

In this article I describe hardware and software that I designed for an IBM PC-compatible computer to acquire real-time data from an external source in a way that still permits the PC to perform analysis and display of the data as it is acquired. A 286-based PC-compatible is an attractive workstation for this kind of development because it is relatively inexpensive, portable, easy to program, and add-on interface hardware can easily be designed for it.

The first problem in a data acquisition system is how to actually acquire the data. Direct memory access (DMA) provides a simple solution from a hardware design standpoint. With DMA, the data from some external device is stuffed directly into the PC memory a byte or a word at a time, without processor intervention. DMA can also be made to work in the other direction: From the PC memory to an external device or even from one area of memory to another. While the transfer of data is taking place, the CPU is free to perform any other tasks it wants to. Its speed will be somewhat degraded because the DMA operation is stealing cycles, holding off the processor for just long enough to transfer the next byte or word. This feature of leaving the processor free allows us to think of this operation as a primitive kind of multitasking: The DMA transfer takes place in the foreground, in real time, while the analysis and display of data takes place in the background, using whatever processor time is left over.



Tom is an associate scientist for Applied Research Corporation, specializing in real-time data acquisition and control and is currently working under contract with the Laboratory for High Energy Astrophysics at NASA's Goddard Space Flight Center. He can be reached at NASA/Goddard Space Flight Center, Code 664, Greenbelt, MD 20771.

PC DMA Architecture

In an IBM PC or compatible, most of the hardware and control signals necessary to carry out a DMA transfer are present on the system board and I/O bus. To take advantage of these "built-in" facilities, you must build a small amount of handshaking circuitry on an interface card that plugs into the PC backplane, then route the data from its external source to this interface card. In keeping with its "open system" philosophy, IBM makes public all the information you need to understand and work with the signals present on its I/O bus (see the References).

On a PC system board, there is a DMA controller, which is an Intel 8237A integrated circuit. A single 8237 controls four separate DMA "channels," numbered 0 through 3. Each can be individually programmed, and all four can be active simultaneously. On a PC, channel 0 is used for refreshing the dynamic memory, and its control signals are not even present on the system bus. Channel 2 is used by the floppy disk, and channel 3 is used by the hard disk, leaving channel 1 available as a spare. Channel 1 can be somewhat oversubscribed, because many add-on peripherals (such as network adapters and tape backup systems) use it.

On an AT, two 8237 chips are present, relieving some of the congestion. The control signals for the additional channels are present only on the ex-

(continued from page 28)
tended AT I/O bus (the short connector in the backplane in front of the longer, standard PC connector). Most commercial add-on boards that use DMA are equipped with switches allowing the selection of a non-interfering channel. However, cards that have only the standard PC connector cannot take advantage of the AT's extra channels. On an AT, channel 2 is used for the floppy, and channel 4 is used for cascading the two controllers. Memory refresh and hard disk transfers are performed without DMA. That leaves channels 0, 1, 3, and 5–7 available for add-on use.

The 8237 DMA controller is a complex chip, occupying a block of 16 addresses in the PC's I/O space. The base address of the primary controller is 0, so I/O ports 0 to F (hex) are used to address it. The secondary (AT only) controller's base address is C0, but it deals in words instead of bytes, and so is addressed on the even-numbered ports in the range C0-DE. Figure 1 shows the I/O port assignments of the 8237 chips.

The following is the sequence of operations involved in setting up and carrying out a DMA transfer. A channel must be selected for use and the hardware and software must both agree on it. The software (writing to the DMA controller) programs the transfer mode (read or write), the starting address of the area in memory to which or from which the data will be transferred, the number of bytes (words, in the case of the secondary controller) to be transferred, and then clears the channel mask bit, enabling the hardware to begin transferring data. The hardware requests a DMA cycle by raising the DMA request line (DRQ) for the particular channel to a high level. The DMA controller responds by lowering the DMA acknowledge line (DACK), which is normally high, for that channel. This signals that the next bus cycle can be used by the hardware to transfer a byte or word of data. If the controller is programmed for a write transfer, when DACK goes low the hardware may place the next value on the data lines. In a read transfer, the I/O bus data lines contain the next value in memory. The controller takes care of everything else: Incrementing the memory location, driving the address lines, and handling the CPU control lines. From a hardware design standpoint, this is a simple interface.

The software operates on a "Ping-Pong" buffer principle

Things become somewhat more complicated if the program desires notification when the DMA transfer is complete. The DMA controller raises a total count signal (TC) along with the last DACK when the programmed count expires. The TC line is available on the I/O bus, and can be wired to one of the interrupt request (IRQ) lines to cause a CPU interrupt when the transfer is complete. The interrupt service routine in software can then program the DMA controller to set up the next transfer, closing the loop and creating a foreground task that will run indefinitely.

PC ctrlr address	AT ctrlr address	Description of register contents
0	C0	Channel 0 base address (write 2 bytes, LSB first).
1	C2	Channel 0 count (write 2 bytes, LSB first).
2	C4	Channel 1 base address
3	C6	Channel 1 count
4	C8	Channel 2 base address
5	CA	Channel 2 count
6	CC	Channel 3 base address
7	CE	Channel 3 count
8	D0	Command/status register (not used in this application)
9	D2	Request register (not used)
A	D4	Single mask register (write only) 7 6 5 4 3 2 1 0 channel select (0-3) mask bit value (0-1)
B	D6	Mode register (write only) 7 6 5 4 3 2 1 0 channel select (0-3) 01=write, 10=read 0=disable autoinitialization, 1=enable 0=disable autoinitialization, 1=decrement 01=single transfer mode
C	D8	Clear byte pointer flip-flop (write only)
D	DA	Master clear (not used)
E	DC	Clear mask register (not used)
F	DE	Write all mask registers (not used)

Figure 1: *8237 DMA controller I/O port assignments*

Building a DMA Interface Circuit

Figure 2 depicts a circuit that performs all the DMA handshaking described earlier. Together with the software supplied in the accompanying listings, it forms the nucleus of a real-time data acquisition system. It was designed to accept data from a D/PAD (a commercial telemetry receiver), but it will work with any data source that can provide a byte at a time and a clock signal when each byte is ready.

At the heart of the circuit is the FIFO (U1 in Figure 2), a chip that implements in hardware the first-in, first-out data structure familiar to most programmers. There are eight parallel data lines coming in to the FIFO, and eight going out. The incoming bytes are written into the FIFO memory by strobing the write clock, and the bytes are read out in the same order as they were written by strobing the read clock. Both of the strobes are active-low pulses. The FIFO acts as an elastic buffer between the data source and the computer: If the computer is busy for some period of time (servicing another interrupt for example), incoming data just gradually fill up the FIFO until the computer gets around to taking them out. If the FIFO ever overflows, data are lost. The depth of the FIFO and the incoming data rate determine the maximum tolerable software delay. The FIFO I used has a

(continued from page 30)
capacity of 1024 bytes.

The FIFO provides an "empty" flag, a signal that is low when the FIFO is empty and high when it contains data. This signal is used directly to drive the channel 1 DMA request line, DRQ1. This means that as long as there is something in the FIFO, there will be a request to transfer data. In single-transfer mode, the DMA controller guarantees that even when DRQ is asserted continuously, the CPU will get every other bus cycle. In other words, we can't shut out the CPU by holding up the request line like this. Assuming that the DMA controller has been correctly programmed by the software, there will be DMA acknowledge signals returned on DACK1. Each DACK, an active-low signal, strobes the read clock on the FIFO, which causes the next byte to appear on the FIFO output lines. The DACK also gates the next chip in line, which is a "tri-state" buffer (U2) whose output is connected to the PC data bus. When the buffer is not enabled (DACK signal high), its output lines are inactive and do not interfere with the PC bus operation. When DACK is low, the buffer drives its 8 bits onto the PC bus, where they are picked up and deposited in memory as directed by the DMA controller.

The number of bytes that are transferred in this manner is dependent on the count programmed into the DMA controller by the software. On the last transfer the DMA controller asserts the TC signal, driving it high. There is only one TC for all the DMA channels, so in order to distinguish it from other DMA operations that may be going on, it is ANDed with the negated DACK signal. The conjunction of these two sets a flip-flop (U3B), whose output goes directly to IRQ3 on the PC bus, causing an interrupt in the CPU. This is the same interrupt used by COM2 if one is present, so another interrupt must be used in case of conflict. To make sure the interrupt is recognized, the interrupt line must be held high until the CPU transfers control to the interrupt service routine. Unfortunately, there is no way to determine automatically that the ISR is executing, so the software must explicitly notify the hardware when it is safe to lower the IRQ. This is done by creating an I/O port.

The first problem in a data acquisition system is how to actually acquire the data. Direct memory access provides a simple solution from a hardware design standpoint

When the software reads or writes from an I/O port, the CPU places the I/O port address on the PC address bus, together with an "address enable" (AEN) strobe and either an IOR or an IOW signal, depending on whether the I/O port was read or written. Normally, any device recognizing its own I/O address uses the next bus cycle to transfer a byte of data, either to or from the CPU, thus satisfying the program's read or write. However, it is not actually necessary for any data to be transferred. The circuit takes advantage of this by using just the fact that an I/O read has occurred on its own address to clear the interrupt request.

The address comparator at U4 raises its output line whenever its hard-wired address appears on the PC bus in conjunction with AEN. For simplicity's sake, the address is only partially decoded, that is, whenever bits 9 – 2 of the address are "11101000" regardless of the rest, the decoder output goes high. Binary 11101000XX is hex 3A0 – 3A3, so any of these addresses will work. (So will 7A0 – 7A3, BA0 – BA3, and FA0 – FA3.) Normally, these addresses are unused in a PC, any other address range can be selected by wiring the appropriate combination of pins to +5V and ground. To finally turn off the flip-flop driving the IRQ, the decoder output is ANDed with IOR. The end result is that the program can read I/O port 3A0 (or any of its equivalents) to clear the interrupt request.

One final bit of logic is used to reset the FIFO to its empty state. This reset is done at the start of data acquisition to clear out any old data and to start filling the FIFO on a "frame" boundary, which marks the beginning of a new block of data. The same I/O port address decode is used in conjunction with IOW this time, to set the flip-flop at U3A, lowering the active-low "reset" signal on the FIFO. The software can thus cause the reset to happen by writing to I/O port 3A0. The flip-flop is cleared at the next "frame sync" pulse from the data source, bringing the FIFO out of reset and allowing it to begin to fill. The software must immediately program the DMA controller to begin a transfer — before the FIFO overflows — to avoid losing data.

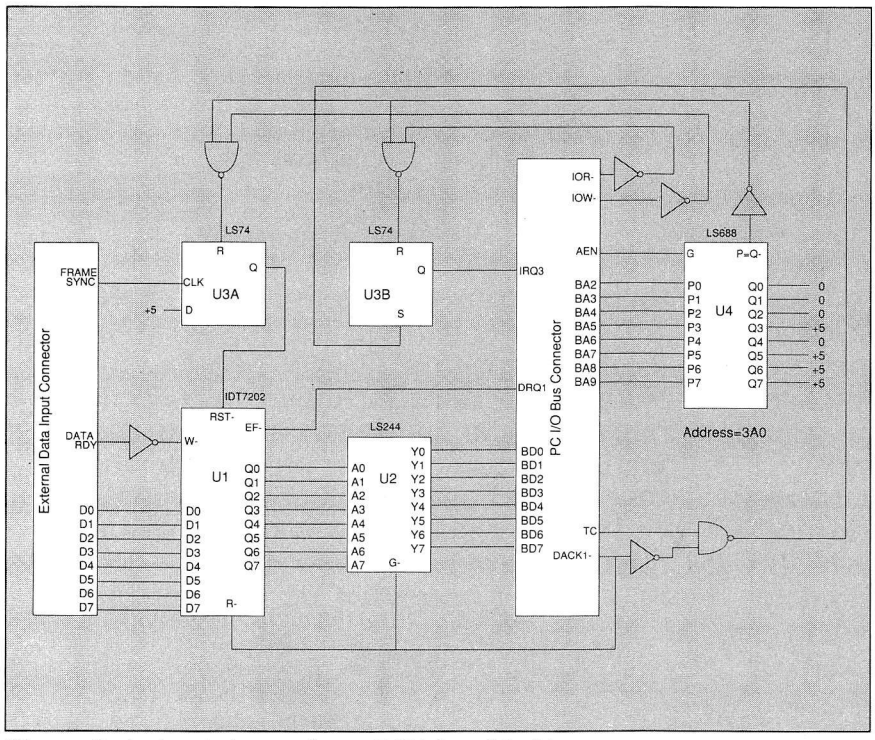


Figure 2: *A circuit that performs DMA handshaking*

Programming the DMA Interface

Now let's turn to the software that makes

Dr. Dobb's JOURNAL
SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

PUBLISHER *Peter Hutchinson*

EDITORIAL
EDITOR-IN-CHIEF *Jonathan Erickson*
MANAGING EDITOR *Monica E. Berg*
TECHNICAL EDITOR *Michael Floyd*
EDITORIAL ASSISTANT *Janna Custer*
CONTRIBUTING EDITORS *Al Stevens, Jeff Duntemann, Martin Tracy, David Betz, Tom Genereaux, Andrew Schulman*
COPY EDITORS *Rosalie Cooke, Pamela Dillehay, Nan Fornal*
EDITOR-AT-LARGE *Michael Swaine*

ART/PRODUCTION
ART/PRODUCTION DIRECTOR *Larry L. Clay*
ART DIRECTOR *Michael Hollister*
PRODUCTION SUPERVISOR *Amy Shulman Lesovoy*
TECHNICAL ILLUSTRATOR *Linda Ann Clark*
TYPOGRAPHERS *Teresa Raines, Margaret Anderson, Charlene Carpentier*
COVER PHOTOGRAPHER *Michael Carr*

CIRCULATION
DIRECTOR OF CIRCULATION *Maureen Kaminski*
CIRCULATION MANAGER *Randy Robertson*
CIRCULATION PLANNING MANAGER *Manny Sawit*
DIRECT MARKETING MANAGER *Andrea Weingart*
NEWSSTAND MANAGER *Sarah Forsman*
DIRECT MARKETING COORDINATOR *Francesca Davies*
PROMOTION COORDINATOR *Pam Moore*
FULFILLMENT COORDINATOR *Anne Jean*

ADMINISTRATION
VICE PRESIDENT OF FINANCE *Kate Deschamps*
CONTROLLER *Mary Collopy*
CREDIT MANAGER *Betty Arsene*
ACCOUNTING SUPERVISOR *Renate Kernke*
ACCOUNTS RECEIVABLE *Wendy Ho*
ACCOUNTS PAYABLE *LuAnn Rocklewitz*

MARKETING/ADVERTISING
DIRECTOR OF SALES AND MARKETING *Karla Spormann*
ADVERTISING COORDINATOR *Laura Stack Pullen*
MARKETING ASSISTANT *Sara Noah Ruddy*
ACCOUNT MANAGERS *see page 160*

M&T PUBLISHING INC.
CHAIRMAN OF THE BOARD *Otmar Weber*
DIRECTOR *C. F. von Quadt*
PRESIDENT *Laird Foshay*
VICE PRESIDENT OF PUBLISHING *William P. Howard*
VICE PRESIDENT/GROUP PUBLISHER *Randall L. Stickrod*

DR. DOBB'S JOURNAL (USPS 307690) is published monthly, except semimonthly in December, by M&T Publishing, Inc., 501 Galveston Dr., Redwood City, CA 94063; 415-366-3600. Second-class postage paid at Redwood City and at additional entry points.

ARTICLE SUBMISSIONS: Send manuscripts and disk (with article and listings) to the editorial assistant 415-366-3600.

***DDJ* ON COMPUSERVE:** Type GO DDJ.

***DDJ* LISTING SERVICE:** 603-882-1599. Supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Type *listings* (use lowercase) at the login prompt.

SUBSCRIPTION: $29.97 for 1 year; $56.97 for 2 years. Foreign orders must be prepaid, including the additional postage (air or surface) in U.S. funds drawn on a U.S. bank. Add $13 for surface mail to all addresses out of the U.S.; add $33 for airmail to Canada and Mexico; or $32 for airlift to all other countries.

POSTMASTER: Send address changes to *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. **ISSN 1044-789X**

CUSTOMER SERVICE: For subscription orders and changes of address call toll-free 800-456-1215 or write *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. For subscription questions outside the U.S. call 1-303-447-9330. For book/software orders call 800-533-4372 (in California 800-356-2002).

FOREIGN NEWSSTAND DISTRIBUTOR: Worldwide Media Service Inc., 115 E. 23rd St., New York, New York 10010; 212-420-0588 FAX 212-420-1265.

Entire contents copyright ©1990 by M&T Publishing, Inc., unless otherwise noted on specific articles. All rights reserved.

The Audit Bureau

(continued from page 32)

this system run. It is contained in two files, dma.c (Listing One, page 94) and test.c (Listing Two, page 96). The first file is a subroutine package containing all of the subroutines to set up and run the DMA hardware, and the second file is a sample main program to show how the subroutines are called.

The software operates on a "Ping-Pong" buffer principle. Two buffers are allocated at run time. While one buffer is being filled by DMA, the other buffer's contents are available to the program for whatever data processing is desired. When the transfer into the first buffer is complete, the buffers are swapped and a new transfer is started. The size of the buffers is chosen so that there is enough time for the program to do all the processing it needs to do in between buffer swaps. At a medium data rate of say, 64K bits per second, there are eight seconds of data in a 64K byte buffer.

Although a lot of processing can be done in eight seconds, it is not an infinite amount of time. If a program needs to do some heavy data crunching, its first action, when buffers are swapped, should be to copy some or all of the new buffers into a spare area in memory and work on it there. The data acquisition is entirely interrupt-driven, and once started, it runs by itself, alternately filling and swapping buffers like clockwork. It can be thought of as the higher priority "foreground" task, with the data processing filling up the time as the "background" task.

The main program calls *alloc_ dma_ buf( )* to allocate the Ping-Pong buffers. The buffer size, in bytes, is specified in the global variable *buf_size*. Because of the way the DMA controller works, neither buffer is allowed to straddle a physical 64K byte "page" boundary in memory. Another way to say this is that each DMA buffer must lie entirely within the range X0000-XFFFF (hex) where X identifies one of the ten

Channel Number	Page Register Address (hex)
0	87
1	83
2	81
3	82
4	8F
5	8B
6	89
7	8A

Figure 3: *DMA page register assignments*

I designed a more complex software package to receive telemetry from a NASA experiment flown on a sounding rocket from Woomera, Australia in March 1988

64K pages in the 640K base memory of the PC. This requirement is difficult to meet and *alloc_dma_buf( )* does it by using the DOS memory allocation functions to grab all available memory, then fitting the two buffers into the first pages large enough to contain them. Of course, the maximum-length DMA transfer this scheme can handle is 64K bytes.

The program then calls *dma_setup( )* to prepare the DMA controller and enable the interrupt service routine. The channel number to be used by the hardware (channel 1 in the example) is specified in the global variable *dma_ chan*, and the interrupt number in *dma_ irq*. *dma_setup( )* locates a number of registers (port addresses) that will be used later on. A single DMA controller can only handle data in 64K chunks (bytes or words, depending on which controller). To gain access to the full 640K address space, the upper 4 bits of the transfer address will be placed in a page register. There is one page register for each DMA channel, allocated as shown in Figure 3. The correct page register address is selected by *dma_setup( )*, as well as the base, count, mask, and mode register addresses from Figure 1.

The next step uses an undocumented feature of DOS. Because DOS is not reentrant, care must be taken in calling DOS functions from an interrupt service routine (it may have interrupted a DOS operation in progress). The undocumented DOS function 34 (hex) returns in the register pair ES:BX, the

(continued from page 34)
address of its critical section flag. When the byte at this address is zero, it is safe to call any DOS function. When the byte is non-zero, DOS is executing a critical section of code and may not be reentered. Function 34 works in DOS, Versions 2 and 3; I have not tested it in Version 4. In *dma_setup( )*, the address of this flag is found and saved for later use.

The final step in *dma_setup( )* is to place the address of the interrupt service routine *dma_isr( )*, in the specified vector, and enable the interrupt by clearing the corresponding mask bit in the interrupt controller.

Starting the Data Acquisition Process

To start up the data acquisition, the program calls *start_dma( )*. Two arguments are passed: The buffer address and the number of bytes to transfer. *start_dma( )* programs the DMA controller with the buffer address and count, using the register addresses computed above in *dma_setup( )*. The 8237 is programmed 1 byte at a time, the least significant byte first. To ensure the correct order, a register on the 8237 resets its internal byte flip-flop to its initial state, but it is a safe assumption that in a PC this flip-flop is already clear so the LSB can be written first.

First, the mask bit on the 8237 for the selected channel is set, disabling DMA on this channel while the 8237 is being programmed. Next, the buffer address is split into 3 bytes by one of two methods, as illustrated in Figure 4. The method used depends on whether the channel number is located on the byte-oriented or the word-oriented controller. The lower 8 bits and the middle 8 bits are written to the 8237's base address register in two successive operations. The high bits are written to the page register. Next, the byte count is divided by two for word-oriented channels, then decremented and output to the 8237's count register in two writes. The reason it is decremented is that the terminal count is reached when the count remaining goes from 0 to FFFF. Finally, the channel mask bit is cleared, allowing DMA to begin.

When the last byte or word is transferred, the interface card causes a CPU interrupt. Because the address of *dma_isr( )* was placed in the corresponding interrupt vector, this routine is entered as soon as the interrupt occurs. By declaring the function with type interrupt, the usual C entry code is replaced by interrupt entry code: The registers are saved and the correct data segment is loaded. The function returns by restoring the registers and issuing an *IRET* instruction. The stack segment may not be the same as the data segment as would be usual in small model code, so pointers to local variables are not allowed in an interrupt context.

In *dma_isr( )* the variable *curr_buf* is set to point to the buffer that was just filled. The main program can use this address to gain access to the data. The buffer index is toggled to swap buffers and *start_dma( )* is called, passing the address of the next buffer. Thus, *dma_isr( )* propagates its own execution because when the next buffer fills, the interrupt will occur and *dma_isr( )*

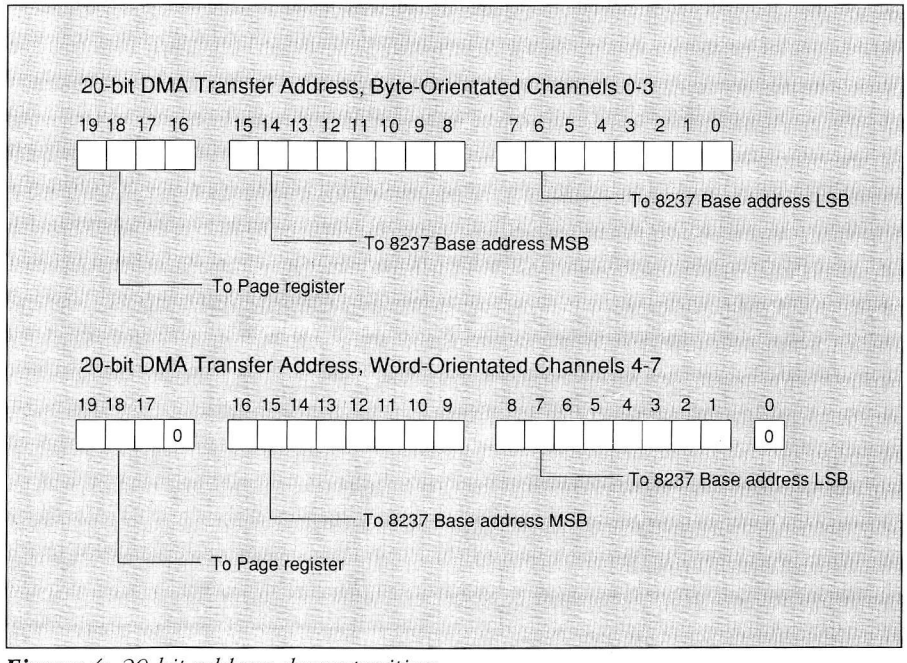


Figure 4: *20-bit address decomposition*

will get called again. Before exiting, the interrupt service routine clears its own interrupt request and outputs the end-of-interrupt signal to the interrupt controller. I declared the function *reset_irq( )* in the calling program rather than here because it is hardware dependent — the functions in dma.c work for any hardware configuration.

If the variable *file_handle* is nonzero, it is assumed that *file_handle* was assigned to a file via a C *open( )* call. *write_buf( )* is then called to write the just filled buffer to the disk file that *file_handle* represents. This is where the DOS critical section flag comes in handy. If the interrupt service routine happened to interrupt DOS at a critical location, the disk write is skipped and the buffer is lost. This will never occur if the background program does not make use of DOS calls, but it provides a simple semaphore when the program must (such as during the opening and closing of files).

Real-Time Applications

I built the circuit using wire-wrap components on a PC prototyping board. It requires only six chips in all, but the FIFO, are standard LS-series parts. I tested it using a simple program on a 12-MHz Compaq 286 computer. The system was able to acquire a continuous stream of data from the D/PAD at rates up to 100K bytes per second, write all the data to disk, and output the results of a minimal error checking algorithm to the screen without dropping any data. Of course, at the highest rate, the available disk storage of 30 Mbytes or so is exhausted in about five minutes!

I designed a more complex software package to receive telemetry from a NASA experiment flown on a sounding rocket from Woomera, Australia in March 1988. With similar DMA interface hardware, the Compaq acquired the telemetry at 25K bytes per second, recorded the full flight's worth of data on disk, and displayed a continuously updating graphical image of the astronomical target. The investment in software and hardware development was very small for a mission of its type, and it has served as a template for future ground support systems design.

References

IBM Corporation, *Technical Reference, Personal Computer AT*, IBM part number 1502243, 1984.

Intel Corporation, *Microsystem Components Handbook*, volume I, Intel part number 230843, 1989.

Eggebrecht, Lewis C. *Interfacing to the IBM Personal Computer*, Howard W. Sams & Company, Indianapolis, Ind. 1983.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 94.)

Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

ZEN for Embedded Systems

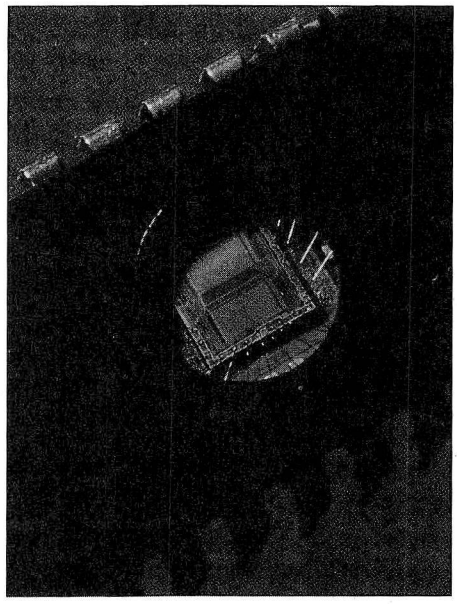
What you see is what you get, with this powerful implementation of Forth

Martin Tracy

ZEN is a tiny Forth with a big purpose: It is a model Forth designed for readability. Have you ever noticed that there are no intermediate Forth programmers? You are either a rank beginner or a black-belt guru. Why? My guess is that after reading Leo Brodie's *Starting Forth* (or my own *Mastering Forth*), you are told that the next step to learning Forth is to read the source code to the language itself. This sounds like a good idea; unfortunately, the source code of a professional Forth system is likely to be tightly packed, handcoded, over-optimized, and generally incomprehensible. If you can make it through all of that, you are in a good position to write a book of your own.

What you need is a Forth that is written mostly in . . . you guessed it! *Dr. Dobb's Journal* has a long tradition of publishing personal implementations of computer languages, from Tiny Basic (January 1976 issue) to Small C (May 1980 issue). So to continue the tradition, we are proud to present ZEN, the tiny Forth.

Programs written in ZEN can be quite small, about half the size of programs written in C. In these days of megabyte memory, is small size important? Yes. Small size often means high speed. On-chip memory generally runs at least twice the speed of off-chip memory. A



68HC11 has only 8K of internal masked ROM. The most popular DSP chip, the TMS32010, has only 4K. Even if the density of on-chip memory doubles each year, three years from now microprocessors will support only 64K of internal ROM.

In recognition of the importance of small programs in embedded applications, ZEN is inherently ROMable and easily fits in an 8K PROM. Not only that, but ZEN is extensible in ROM. How can a program grow in ROM? Assume that ZEN is resident in a single-board computer (SBC) and is talking to a host computer over an RS-232 serial line. Put a RAM chip in an empty ROM socket, and then send source code to ZEN with instructions to compile it to the address occupied by the RAM. ZEN will write the program there but will assume that all references to RAM point to a separate address space, normally the on-chip RAM.

Test the program interactively and, once it works, tell ZEN to read the RAM image and burn it into a PROM. Finally, remove the RAM chip, plug in the PROM, and you're done. The idea of running the development system in the target hardware may seem novel; what it means is that all testing and debugging take place in the target system. The host computer is used only as a terminal and disk server. The chances of a working program not working when you burn the final PROM are utterly remote.

ZEN Internals

Before you study the ZEN implementation (see Listing One, page 98), you must have a fairly good understanding of Forth. For example, you should know that the body of a CODE definition contains machine code. You may wish to treat the 80 or so CODE words in the ZEN kernel as black boxes, but be sure you know what they're supposed to do. Otherwise, be aware that Forth assemblers differ somewhat from classical assemblers. In most Forth assemblers, operands precede operators.

AL 0 [BX] MOV

instead of

Martin is a DDJ *contributing editor and a consultant. He can be reached at 2819 Pinkard Ave., Redondo Beach, CA 90278.*

(continued from page 38)
```
MOV [BX],AL
```

Furthermore, the ZEN assembler uses numeric local labels

```
      DX AX OR
      1 L# JZ
      AX PUSH
1 L:  BX PUSH
```

instead of using

```
      OR AX,DX
      JZ away
      PUSH AX
away  PUSH BX
```

ZEN 1.7 for the 80x86-based IBM PC is a direct-threaded Forth. The SI register points to the next address to execute in a list of compiled addresses. The NEXT interpreter transfers control to the machine code at that address, simultaneously adjusting SI to point to the next address.

```
NEXT    WORD LODS    AX JMP ;
```

The overhead of interpreting a list of ZEN words is therefore two instructions per word. A list of addresses, however, takes far less memory than a series of jumps or calls. In a sense, threaded Forths trade speed for memory. In other words, threaded Forths compile tokens, but the token of a word is simply the address of that word. Microsoft's Compiled Basic and modern "cddr-coded" LISPs have recently copied this method.

Assume that SI points into a Forth definition at a point where it executes DUP. That means that SI points to the compiled address of machine code that duplicates the item on top of the stack.

```
CODE DUP    BX PUSH    NEXT C;
```

The *C;* command is an abbreviation of END-CODE. This implementation of ZEN keeps the top stack item in the BX register. DUP executes in only three instructions — one for PUSH and two more for NEXT. This optimization applies to many other machine-coded primitives, such as DROP and @.

```
CODE DROP    BX POP       NEXT C;
CODE @       0 [BX] BX MOV NEXT C;
```

ZEN is a small model (64K) Forth. All four segment registers contain the same segment address. The origins and sizes of the separate RAM and ROM address spaces are specified on block 1.

ZEN Word Set

The ANSI Forth standardization effort is well under way. The current document, ANS X3J14 BASIS 9, is available for $6 from the X3J14 Secretariat (c/o FORTH, Inc., 111 N. Sepulveda Blvd., Manhattan Beach, CA 90266). The BASIS is modified at each meeting until it becomes the draft proposal that, after a period of public review and revision, becomes the ANSI Forth Standard. ZEN 1.7 is an unofficial implementation of BASIS 7 and provides all required words. It supports the double number, file access, and BLOCK extensions.

The stack manipulation words found in ZEN are:

```
DUP DROP SWAP OVER
                          ROT ?DUP
 2DUP 2DROP 2SWAP 2OVER 2ROT
>R R@ R>  2>R 2R>
NIP TUCK PICK ROLL
```

ROLL is defined in high-level because it cannot be efficiently mapped into machine code. You will find that 2>R and 2R> obviate almost any need for ROLL. In case you are not familiar with NIP and TUCK, they are equivalent to the phrases SWAP DROP and SWAP OVER, respectively. NIP is especially handy and is a single instruction on the Harris RTX 2000 Forth Processor. It only takes a single instruction (plus NEXT) on the 80x86 as well.

The single-precision arithmetic operators form this set:

```
+  – * / MOD /MOD */
1+ 1/  2* 2/
NEGATE ABS   MAX MIN
```

1+ and 1– increment and decrement, respectively. 2* and 2/ shift once right and left, arithmetically. Notice that 2+ is not available. Use CELL+ instead to move to the next cell in an array. Use CELLS instead of 2* to change cells into bytes, as in:

```
1024 CELLS ALLOT.
```

Many single-precision operators have double-number equivalents.

```
D+ D- D2* D2/
DNEGATE DABS DMAX DMIN
```

Multiplication and division generally mix precisions to avoid calculations with unnecessary accuracy.

```
UM* UM/MOD   */MOD M* M/MOD
M+  S>D  D>S
```

A full complement of logical and comparison operators exists.

```
AND OR XOR NOT
0< 0= 0>  < = >  U<  WITHIN
D0= D< D=  TRUE 0 1
```

WITHIN is especially powerful because it compares within in a circular number space. It can be used to control the duration of an event by comparing the current value of a self-incrementing counter to some future value. All other comparison operators can be based on WITHIN.

These are the memory access primitives:

```
@ ! C@ C! 2@ 2! D@ D!
CMOVE CMOVE>  MOVE  +!
```

D@ and D! are equivalent to 2@ and 2! except that the order of cell storage is not specified. MOVE is a smart CMOVE that moves without overlapping bytes. It cannot be used to fill memory with a character.

Here are the flow-of-control operators:

```
BEGIN WHILE REPEAT UNTIL AGAIN
IF ELSE THEN   DO LOOP +LOOP
I J  LEAVE  UNDO  EXIT
EXECUTE  @EXECUTE
```

UNDO removes one level of DO . . . LOOP support from the return stack, allowing EXIT to execute within the loop. UNDO EXIT is the preferred method for leaving a word from inside a loop. @EXECUTE is exactly equivalent to @ EXECUTE, but is much faster.

These are the number conversion and formatting commands:

```
BASE DECIMAL HEX
CONVERT VAL?  PAD
<#  #  #S  SIGN HOLD  #>
. U. D.  D.  R
```

Output conversion and formatting takes place at PAD 1– and downwards, leaving PAD itself free for applications. VAL? converts strings to numbers, and BASIS 6 defines numeric, string, and character literals.

```
[ASCII] ASCII  "ccc" LITERAL
```

ZEN adds DLITERAL, the double-number equivalent of LITERAL. ASCII and [ASCII] make the following character into an ASCII literal inside and outside

(continued from page 42)
a definition, respectively. The string literal operator " (quote) works only inside a definition and accepts the following string of characters, up to the next ", as a string literal.

```
COUNT SKIP SCAN /STRING
FILL BLANK ERASE -TRAILING
STRING ." EVALUATE
```

SKIP SCAN and /STRING are natural factors of WORD. SKIP and SCAN closely map the 80x86 string operator SCAS. /STRING is a generic substring operator; it can be used to select any part of a string. STRING is the fundamental string compiler and is used by the "and ." commands.

EVALUATE is the most powerful string operator: It interprets any string. EVALUATE is used for forward referencing, macro definition, and any other instance of late binding. For example, coldstart uses the phrase READY EVALUATE for device initialization. Suppose you add a device that requires its own initialization:

```
: READY ( new initialization) READY;
```

READY is redefined to include the new initialization. READY EVALUATE executes this new version, which in turn executes the previous version.

The interpreter level uses these words:

```
BLK >IN TIB #TIB SPAN STATE
FORTH CONTEXT ' FIND WORD
CR TYPE EXPECT KEYS KEY?
KEY EMIT PAGE MARK TAB
BL SPACE SPACES .( ( DEPTH
```

Most Forths have words such as KEY?, which is true if a key is available; PAGE, which clears a screen or page; MARK, which TYPEs in emphasized mode; and TAB, which repositions the cursor to a given X Y coordinate. None of these words have been standardized. In addition, ZEN supports KEYS, which accepts characters without echoing or editing. In general, if the input stream is coming from a human being, use EXPECT; otherwise, use KEYS.

The compiler layer adds these words:

```
CURRENT DEFINITIONS [ ] ['] , C,
COMPILE [COMPILE] IMMEDIATE
                        RECURSE
```

RECURSE allows a definition to reference itself. Remember, though, that the size of the return stack is likely to be limited. The following defining words are provided:

```
CREATE VARIABLE CONSTANT : ;
2VARIABLE 2CONSTANT
                        VOCABULARY
```

Two other defining words are used to build ZEN: USER and XFER. USER creates a user variable, whose address is based on its offset from the address in the pseudoregister *u*. A user variable is private to each task in a multitasking environment. XFER creates a transfer command, whose action is specified by its offset in an execution vector pointed to by the user variable *x*. By changing this execution vector, input and output can be redirected to different devices or files. Because *x* is a user variable, each device can be controlled by a different task, so a background task can spool text to the printer while a foreground task paints graphics on a screen.

RAM vs. ROM

The RAM and ROM address spaces in ZEN are kept entirely separate. The dictionary, machine code, and tables are built in ROM, and the data fields of variables and arrays are built in RAM. Each word that refers explicitly to RAM has an equivalent that refers instead to ROM.

RAM	ROM
CREATE	VARIABLE
C, ,ALLOT	
HERETHERE	
DOES>	GOES>
>BODY	>DATA

Let's say you want to make a table of powers of ten:

```
CREATE TENS
1 , 10 , 100 , 1000 , 10000 ,
```

The TENS table will be built in ROM. Suppose you want an array of seven cells:

```
VARIABLE DAYS 6 CELLS ALLOT
```

The DAYS array will be built in RAM. Notice that VARIABLE allocates the first cell. It is not possible to initialize DAYS because anything stored in RAM disappears when the power is turned off. You can use GAP to allocate uninitialized cells in ROM, but you will rarely need to do so.

DOES> refers to the body of a defined word in ROM.

```
: COLOR \ set CREG to the value n.
    CREATE ( n) , DOES> @ CREG ! ;

8 COLOR RED 12 COLOR GREEN
```

Executing RED will store 8 into CREG. DOES>DATA to the data field of a defined word in RAM.

```
: KOUNTER \ self-incrementing object.
VARIABLE DOES> DATA 1 SWAP +! ;

KOUNTER ALPHA KOUNTER BETA
```

Each execution of ALPHA will increment its value. To read this value, you must be able to recover the data field address.

```
' ALPHA >DATA ? ( prints value)
```

Use >BODY to recover the parameter field address of an object in

```
ROM: ' RED >BODY ? ( prints 8)
```

Restart and Error Handling

The restart and error-handling mechanism, designed by Wil Baden, was published in *FORML 1987*. The Forth command-line interpreter is QUIT; it is the default Forth application.

```
: QUIT RESET
   BEGIN CR QUERY interpret OK?
AGAIN ;
```

When Forth begins executing, it goes to ABORT and from there to QUIT. The function of ABORT will be performed by every error recovery sequence. ABORT is the default Forth error restart sequence.

```
: ABORT
BEGIN PRESET QUIT GRIPE AGAIN;
```

In QUIT, the work of clearing the data stack is factored into RESET. The data stack, however, is not completely cleared; one item is left, but this has no impact on existing programs. In ABORT, PRESET clears both stacks all the way to the bottom.

You now have a way to customize the error recovery sequence. In addition, ABORT shows the pattern for dedicating an application: The default application is QUIT, and the default error recovery sequence is GRIPE. GRIPE sounds like TYPE and takes the same parameters, address, and length. It displays the error message provided by ABORT. It may also attempt to display the name of the word being interpreted.

PRESET leaves both stacks empty. Invoking the next word will put the address of the return point on the return stack. RESET leaves that address alone. A word can get back to that location by RESET EXIT. On the return, the parameters for an error message string are assumed.

(continued from page 44)

The default application and default error recovery sequence can be changed, and ABORT" will still work with them.

```
: err RESET ;

: ABORT" [COMPILE] IF [COMPILE]"
  COMPILE err [COMPILE] THEN ;
IMMEDIATE
```

err in the definition of ABORT" cannot be replaced with RESET, but you can build a variable error message string and follow it with RESET EXIT or RESET ; to emulate ABORT".

The following trivial example changes the default application to indent three spaces for every value on the stack before receiving new input. This process is preventative debugging: If you are surprised by the indentation then you are in trouble. Also, the example demonstrates logical structure as you compile a definition from the keyboard.

The error recovery message is changed as well.

```
: OLD-ABORT  ABORT ;

: INDENT DEPTH STATE @ -3 *SPACES;

: SHELL
   BEGIN CR INDENT
      QUERY interpret OK?
   AGAIN ;

: ABORT
   BEGIN   PRESET SHELL
      CR ." Abort: " GRIPE
   AGAIN ;
```

ABORT installs a new application and error handler. OLD-ABORT reinstalls old application and error handler.

With PRESET and RESET you can establish intricate error recovery and restart networks. The following generic QUIT can be used to restart the default application in most implementations. The idea is to return to the position just before you invoke your dedefault application.

```
: QUIT RESET R> CELL - >R ;
```

Mass Storage

ZEN supports both text files and traditional Forth BLOCKs. The text file support includes these words:

```
CREATE-FILE     OPEN-FILE  CLOSE-FILE
DELETE-FILE     RENAME-FILE
READ-FILE       WRITE-FILE
SEEK-FILE       FILESIZE   FILEPOS
READ-LINE       WRITE-CR
IO-RESULT
```

READ-FILE and WRITE-FILE read a sequence of bytes from the current position in the file to and from a buffer in memory. SEEK-FILE and FILEPOS change the position of a file, and FILESIZE returns its size. READ-LINE reads a line of text, and WRITE-CR writes an end-of-line terminator. To write a line of text, use WRITE-FILE followed by WRITE-CR. All file primitives work with a file identifier returned by CREATE-FILE or OPEN-FILE, so multiple files are supported. Error codes, if any, are returned in IO-RESULT.

The BLOCK support includes these words:

```
BLOCK BUFFER   UPDATE
SAVE-BUFFERS FLUSH   LOAD
```

Running ZEN

You can find the source code and executable image of ZEN in the file ZEN190.ARC on CompuServe in the DDJ Forum. ZEN is also available on the GENIE, ECFB, and BIX bulletin boards. Execute the file ZEN.COM and press <RETURN>. You should see the OK message. ZEN has a small set of debugging words you can use to look around:

```
WORDS .S ? DUMP
```

You can compile and test definitions from the keyboard, or type GO to compile the text file KERNEL.SRC. Use any text editor to edit this file. I use the Sidekick pop-up editor.

Use GUARD to protect all words in the dictionary from FORGET. EMPTY restores the dictionary to the previous GUARDed word set.

I hope you enjoy ZEN and find it useful. Please send all comments and criticisms to me.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listing begins on page 98.)

Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

Error Message Management

Automating error message documentation

Rohan T. Douglas

One of the more overlooked problems programmers face during the software development life cycle is error message management. For one thing, a list of error messages must usually be provided in the user manual, along with detailed descriptions. At one stage or another, many of us may have resorted to a combination of *grep* and our favorite text editor to produce such a list. For larger projects, however, this approach becomes unmanageable.

The purpose of this article is to present an approach which is simple to implement and use. It offers some significant advantages over more traditional approaches to the problem. In particular, this approach automates the error message documentation process, guaranteeing consistency between the product and documentation, and adds the ability to generate error messages in one of several written languages. For that matter, there's no reason this scheme couldn't be adapted for on-screen menus, prompts, and dialog boxes.

Rohan Douglas is an associate director of Giltnet Limited and has been an architect of a number of real-time analytic products for the global fixed interest markets including the product Tactician. Rohan can be reached at G.P.O. Box 2424, Sydney, N.S.W. 2001, Australia or by FAX (2)-262-5357 (Australia).

An Error Message Data Base

One traditional approach to the problem of error message management is to maintain a separate data base of error messages which are referenced within the source code by calling some function with an identifying index. This approach, however, presents several problems.

For readability, a description of each error must also be included in the source. The problems of maintaining two copies of the same information quickly become apparent. Cross-checking the source against the error message data base is a problem to be avoided.

Maintaining a distinct error message data base is also fairly cumbersome. In the worst case, having come across an error condition while editing source code, we have to exit the source editor to call up our error message data base editor. Other issues to consider involve multiple write access to this error message data base not necessary.

The Solution

A more natural approach is to retain the error messages within the source code. This avoids the problems listed above, but how do we generate a detailed listing of error messages along with descriptions? The solution takes the form of a source processor.

The source processor takes a completed source program and scans for error messages. By embedding a full description of each error message within the source code, we can automatically generate a list of error messages along with descriptions suitable for inclusion in a user manual.

If we extend this concept further, we can combine the facilities of the C preprocessor with the source processor, allowing us to automatically generate an external data base of error messages. This allows not only conversion to foreign languages but also such final touches as a spell-check for the production version of the product.

The source processor approach can encourage more readable code by documenting error messages thoroughly within the source code, automating the generation of error message descriptions, allowing fine-tuning of the error messages for the production version without touching the source code, and allowing multi-language error messages using the same program executable. Another advantage is that the source processor approach does not impede the flow of the development process. The source processor need only be run on the production version of the source.

I have used the source processor approach on a number of real-time financial systems, each around 300,000 lines of C, and it has proven both flexible and practical.

Implementation

The implementation essentially consists

(continued on page 51)

(continued from page 48)
of two components. The first is a macro, *error( )*, which replaces the error message in the source with a call to an error-handling function *_error( )*. This error-handling function is passed the file name and line number which are used to look up an error data base file. The definition for the error macro is shown in Listing One, page 108, and the definition of the error-handling function is shown in Listing Two, page 108.

The implementation shown is fairly simple. More than likely you will want the error-handling function to pop up some window rather than use *printf( )*. The ability to pass additional information to the error function may also be required. These are fairly straightforward extensions.

The second component of the error handling system is the source processor. The function of the source processor is to scan through the source and extract error messages and accompanying descriptions. The file and line numbers of each error message are noted and two files are produced. The first is a listing of the error messages with descriptions suitable for inclusion in a user manual. The second is a data base of error messages indexed by their respective file and line numbers. Figure 1 shows a sample source file with an embedded error message and description. Figure 2 shows the resulting description for the user manual while Figure 3 shows the message data base. Figure 4 shows the results of the sample program.

An AWK listing of the source processor is shown in Listing Three, page 108. Some assumptions have been made about the format of the embedded error message comments. This listing is intended as a starting point rather than a definitive implementation.

```
#include <stdio.h>
#include "error.h"

main(int argc, char *argv[ ])
{
    if (argc == 1)
        error("Missing parameters");
        /* No parameters have been passed
         * to this test program. This error message
         * will be displayed as an example if no
         * parameters are passed to this test
         * program.
         */
    exit(0);
}
```

Figure 1: *Sample source file with embedded error message and description*

```
Missing parameters:
        No parameters have been passed
        to this test program. This error mes-
        sage will be displayed as an ex-
        ample if no parameters are passed
        to this test program.
```

Figure 2: *The user manual list resulting from Figure 1*

```
test:8 Missing parameters
```

Figure 3: *The message data base resulting from Figure 1*

```
Missing parameters [test:8]
```

Figure 4: *The results of the sample program*

Conclusion

An error message source processor provides a flexible and effective mechanism for controlling error messages. Embedded error messages along with descriptions are extracted from the source code to create an external error message data base for a production version of the product and error message listing suitable for inclusion in a user manual. The benefits of retaining the error messages within the source code are maintained while the benefits of an external error message data base are obtained for the production version of the product.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 108.)

Vote for your favorite feature/article.
Circle Reader Service **No. 4.**

S-CODER for Data Encryption

A secure encryption method need not be computationally intensive

Robert B. Stout

Although not a new topic, surprisingly little has been done to secure data in the microcomputer world. The problem is that most encryption systems were developed for high-powered mainframe and super computers that can handle the computationally intensive calculations required for such encryption schemes. Most small computers, on the other hand, cannot reasonably implement algorithms set forth by agencies such as the U.S. National Bureau of Standards.

On the upside, however, most small computer users don't need the heavy security required by such government agencies. And while simple encryption schemes can be easily broken, there is a way to get the best of both worlds. This article presents a new algorithm to ease the production of secure data systems. Although written in ANSI C, the algorithm as presented is adaptable to most high-level languages, assembly, and even hardware implementations. Although usable "as-is," it can also be used as a building block for enhanced security applications.

Bob is the president of MicroFirm, a company specializing in utilities for small computers. He can be reached at P.O. Box 428, Alief, TX 77411.

A Quick Tour

A typical data encryption system consists of three basic elements: The unencrypted data, referred to as the "plaintext;" an encryption key; and an encryption method (the encryption engine).

The output of the encryption process is called the "ciphertext." Decryption uses three corresponding elements, except that the input is the ciphertext and the output is the reconstructed plaintext.

"Cryptanalysis" is the science of breaking ciphers based on searching for specific patterns in the ciphertext. If the plaintext is unknown, the patterns represent a best guess as to the partial contents of the plaintext. Even better is a known plaintext attack where a known message has been encrypted so both the plaintext and the ciphertext are known.

Most common small computer data encryption schemes are based on either the exclusive ORing of a given text string key with the plaintext (a polyalphabetic substitution cipher as shown in Figure 1), or a software implementation of the Data Encryption Standard (DES) as published by the U.S. National Bureau of Standards. The former is not particularly secure, while the DES is computationally intensive. The exclusive OR method would be perfectly secure if the key were longer than the plaintext (called a "one-time" key) and if the key were perfectly random.

Despite the high-tech appeal of DES

```
/*****************************************************************************/
/*  Simple encrypt/decrypt function using exclusive-ORing                    */
/* NOTE: This is included for demonstration only! Data encrypted             */
/* with this code will be subject to simple cryptanalysis.                   */
/*****************************************************************************/
 char *cryptext;        /* The encryption/decryption key                     */
void crypt(char *buf)
{
                int crypt_ptr = 0; /* Circular pointer to elements of key   */
        *buf ^= cryptext[crypt_ptr];
        if (++crypt_ptr >= strlen(cryptext))
            crypt_ptr = 0;
}
```

Figure 1: *A polyalphabetic substitution cipher*

(continued from page 52)
and public key encryption systems, these systems were developed both in response to, and dependent upon, the general availability of powerful specialized hardware. Keys are typically large numbers and are difficult to memorize. DES requires 16 iterations of nested substitution and permutation ciphers. RSA public key systems involve raising 200 digit numbers to 200 digit powers. Most PC users simply want a secure means of quickly making files unreadable by unauthorized personnel.

The S-CODER Engine

The S-CODER algorithm is the core of an encryption "engine" (an S-Box, or substitution cipher module), which is a variant of the exclusive OR method. It differs primarily in that it "scrambles" the key on-the-fly (characteristic of a stream cipher) to approach the security level of using a one-time random key. The complete algorithm is implemented in Figure 2.

To remain secure from known plaintext attacks and minimize the impact of poorly selected keys, the exclusive OR product is sufficiently complex to thoroughly hide the encryption key, even when encrypting strings of identical characters. By modifying the key according to known rules, the key effectively becomes an array of seed values for a complex random number generator. It is these random numbers, rather than the key text itself, which are exclusive ORed with the plaintext.

Key distribution is a historical problem for any encryption scheme

S-CODER requires that three global variables are set up by the calling program. The encryption key goes in cryptext, an index into the key goes into *crypt_ptr*, and the length of the key is set in *crypt_length*. Although *crypt_ptr* is usually set to point to the first character of the key, this may be changed in more secure S-CODER applications. The only reason for passing *crypt_length* in a variable is to avoid

```
/*****************************************************************************/
/*      S-CODER — Encrypt/decrypt data                                       */
/*      Copyright 1987 – 1989 by Robert B. Stout dba MicroFirm               */
/*      Originally written by Bob Stout with modifications                   */
/*      suggested by Mike Smedley.                                           */
/*      This code may be used freely in any program for any                  */
/*      application, personal or commercial.                                 */
/*      Current commercial availability:                                     */
/*      1. MicroFirm Toolkit ver 3.00: LYNX and CRYPT utilities              */
/*      2. CXL libraries (MSC, TC, ZTC/C++, PC):fcrypt( )                    */
/*      dedicated file encryption function                                   */
/*      3. SMTC & MFLZ libraries: crypt( ) function                          */
/*****************************************************************************/
char *cryptext;         /* The actual encryption/decryption key */
int    crypt_ptr = 0;   /* Circular pointer to elements of key */
int    crypt_length;    /* Set externally to strlen(cryptext) *
/* NOTES:  cryptext should be set and qualified (to something over 5 – 6 chars, minimum)
           by the calling program, which should also set crypt_ptr in the range of 0 to
           strlen(cryptext) before each use. If crypt( ) is used to encrypt several buffers,
           cryptext should be reloaded and crypt_ptr reset before each buffer is en-
           crypted. The encryption is both reversible — to decrypt data, pass it back
           through crypt( ) using the original key and original initial value of crypt_ptr —
           and multiple passes are commutative. */
/**** Encrypt/decrypt buffer datum ****************************/
void crypt(char *buf)
{
    *buf ^= cryptext[crypt_ptr] ^ (cryptext[0] * crypt_ptr);
    cryptext[crypt_ptr] += ((crypt_ptr < (crypt_length - 1)) ?
        cryptext[crypt_ptr + 1] : cryptext[0]);
    if (!cryptext[crypt_ptr])
        cryptext[crypt_ptr] += 1;
    if (++crypt_ptr >= crypt_length)
        crypt_ptr = 0;
}
```

Figure 2: *The S-CODER algorithm*

(continued from page 54)
repetitively calling the *strlen( )* library function. An important point to note is that because the key is scrambled during encryption, to encrypt multiple texts using the same key requires the unaltered key to be reloaded into cryptext before each use, and that *crypt_ptr* be preset to its initial value.

In examining the S-CODER code, note that both the plaintext and the key are masked by including the exclusive OR of the product of *crypt_ptr* (changes each character) with the first character of the key (changes each pass). Because the length of the key is unknown and the product is implicitly modulo 256, this simple scheme renders cryptanalysis extremely difficult by searching for key-length cycles. This is especially true if we further assume that *crypt_ptr* may be preset to some random starting point within the key.

Despite the high-tech appeal of DES and public-key encryption systems, these systems were developed both in response to, and dependent upon, the general availability of powerful specialized hardware

As each character of the key is used, it is modified by summing it with the character, which is logically to its right, thus treating the key data as a circular buffer. Again, this is implicitly a modulo 256 operation. Because the unique character zero or NULL is reserved in C to denote the end of a string, zeros are converted to ones to preserve the key string length and to save the degenerative effects of zero propagation through the mathematical operations on the key.

S-CODER Applications

Listing One (page 110) implements a simple, single file encryption utility using the S-CODER algorithm. For simplicity, *crypt_ptr* is initialized to zero, and no key qualification or validation is performed. Even this simple application has proven resistant to unknown plaintext attacks.

Listing Two (page 110) demonstrates the use of the S-CODER algorithm as a stream (*stdin* to *stdout*) encryptor. It includes a call to a *setraw( )* function, which will be different for different compilers. This is necessary to force the *stdin* and *stdout* streams into "raw" or binary mode for the duration of the program's execution. Listing Three (page 110) demonstrates a simple *setraw( )* function for the Zortech C and C++ compilers. Microsoft and Turbo C users may use the *setraw( )* function in their compilers.

Listing Four (page 110) demonstrates a more secure implementation and introduces a *cryptqual( )* function to assure that the encryption key contains at least six distinct characters. The S-CODER engine is embedded in an encryption algorithm, where data are read into and out of a 16K data buffer as linear arrays but encrypted as two-dimensional 128 × 128 columnar character arrays with random padding of the final buffer (a modified 16K block transposition cipher).

Also note that *crypt_ptr* is preset to a computed value rather than the default of zero. This computed value includes the length of the encrypted file, which is written along with the ciphertext. Although the S-CODER algorithm effectively masks the key length, these techniques serve to eliminate residual weaknesses when faced with a known plaintext attack. Using the basic S-CODER engine as a building block, this example suggests some ways to build more secure specific applications.

Listing Five (page 111) shows a simple test program to calculate statistics on test files. The coefficient of variation of these examples asymptotically approaches zero as the size of the file increases. For 100K test files, it has typically been down around five percent.

Summary

There are several important issues that S-CODER doesn't address. Key distribution is a historical problem for any encryption scheme. Public key encryption and key-sharing schemes address this issue, but at the expense of hard to memorize keys, authentication problems, and computational complexity. S-CODER's operation is symmetrical and also commutative when used with multiple keys. For example, if data are doubly encrypted with separate keys, the resulting ciphertext may be decrypted by passing it through an S-CODER engine using the original keys in any order. This suggests the use of

(continued from page 56)
a hierarchical key distribution scheme.

No data encryption system is unbreakable. But it's important to remember that, for hundreds of years, there have existed relatively simple ciphers, which couldn't routinely be broken before the invention of supercomputers. If your adversaries can't afford anything more powerful than a 80386-based PC, your data would probably be safe with any of these methods. Also, in the world of industrial espionage, it's usually cheaper for an adversary to buy your employees than the computing power to crack your data when encrypted with tools such as S-CODER.

The real purpose of encryption is to make cryptanalysis prohibitively expensive, either in terms of cost or time. A further fact of life is that any encryption process that runs acceptably fast on a small computer is vulnerable to cryptanalysis on much larger and faster machines. The S-CODER algorithm is an effective compromise — easy to use, reasonably fast on small machines, yet offering effective levels of data security.

Notes

Computer Networks by Andrew S. Tannenbaum, ISBN 0-13-162959-X, Prentice Hall.

An Introduction to Cryptology by Henk C.A. van Tilborg, ISBN 0-89838-271-8, Kluwer Academic Publishers.

Security in Computing by Charles P. Pfleeger, ISBN 0-13-798943-1, Prentice Hall.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 110.)

Vote for your favorite feature/article.
Circle Reader Service **No. 6.**

Parametric Circles

Faster circle drawing algorithms can produce time savings that are an order of magnitude faster

Robert Zigon

As graphical PCs become increasingly popular, end-users continue to demand faster response rates until their problem can be modeled interactively in real time. The obvious solution to the problem of minimizing calculation and regeneration time is faster hardware, but this is the brute force solution. Let's instead look at producing better software through clever algorithms. This solution can produce time savings that are an order of magnitude faster. This article describes an algorithm for efficient circle generation and shows how this is achievable through a modest amount of mental gymnastics.

In computer graphics, circle (and ellipse) generation is an operation as fundamental as line drawing and point plotting. The obvious approach to generating the (X_k,Y_k) pairs needed to describe the circumference of the circle is shown in the equation in Figure 1. However, this implicit nonparametric representation of the circle has three problems.

First, a circle has multiple Y values for a given X value. This necessitates solving the equation in Figure 1 for Y_k to produce the equation in Figure 2. Because the goal is to produce a fast algorithm, it is best to avoid the extraction of the roots. The second problem deals primarily with the aesthetics of the curve. When the X_ks are evenly spaced, the results are poor due to the uneven distribution of points along the length of the curve in Figure 3.

Robert is a senior software engineer for International Laser Machines and can be reached at 4645 Orlando Ct., Indianapolis, IN 46208.

Finally, implicit nonparametric curves are axis-dependent. The implication here is that the choice of coordinate systems can affect the ease of calculation. This becomes especially apparent when the end point of a curve has a vertical slope relative to the chosen coordinate system. These problems are why we will switch to parametric representations.

The parametric form of each point on the circumference of the circle can be represented as two functions of one variable. There are many possible choices for the functions and parameters. The equation representing Figure 3 can be converted to its normalized parametric equivalent according to the equation in Figure 4. However, as previously mentioned, Figure 3 was an example of a computationally inefficient and aesthetically displeasing portion of a circle. This leads us to search for a different parameter. As such, we will look to polar representations with the parameter θ. Figure 5 describes our new parametric form of a circle.

$$X_k^2+Y_k^2 = R^2$$

Figure 1: *Equation to generate pairs that describe a circle*

$$Y_k=\sqrt{R^2-X_k^2}$$

Figure 2: *To solve the equation in Figure 1, it's necessary to extract the roots*

At first glance, Figure 5 might not seem like much of an improvement over the equation in Figures 2 or 4. The evaluation of the sine and cosine functions can be just as costly from a computational standpoint as the extraction of the square root. We could precompute a table of sine values, but this would limit us to a fixed number of points. The way to calculate the (X_k,Y_k) pairs is by rearranging the equation in Figure 5. Observe that θ_ks used in Figure 5 can be generated from the previous value via the relation $\theta_{k+1} = \theta_k +$

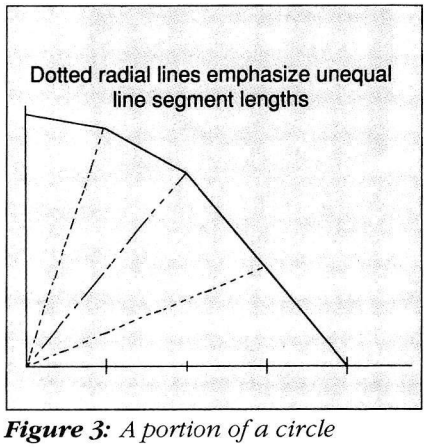


Figure 3: *A portion of a circle*

$$Y(t) = \sqrt{1 - t^2} \qquad 0 \le t \le 1$$
$$X(t) = t$$

Figure 4: *The normalized parametric equivalent of the circle in Figure 3*

(continued from page 60)

$\Delta\theta$. If we take the cosine of both sides of the previous equation, we have $\cos(\theta_{k+1}) = \cos(\theta_k + \Delta\theta)$. The right side of this equation can be expanded by using the trigonometric identity for the cosine of an angle θ and some $\Delta\theta$. The identity is shown in Figure 6.

If you now multiply by R, the radius of the circle, you will essentially have the equation in Figure 5. However, because the right side has been expanded, according to the equation in Figure 6, what you will actually be left with is the equation in Figure 7. The significance of this expansion can be found in the third line of the equation for both X_{k+1} and Y_{k+1}. The use of the trig identity shows that X_{k+1} is partially described in terms of X_k and Y_k, leaving us with a recurrence relation. Notice that the X_k and Y_k values are multiplied by the sine and cosine of $\Delta\theta$, a numeric constant! The implication of these algebraic gyrations is that the calculation of the equation in Figure 7 is simply the sum of the product of two sets of numbers.

Please note that Figure 7 will generate the points on the circumference of a circle centered at the origin. In general, if the circle is centered at the point (a,b), then the equation in Figure 8 will be of greater interest.

$$X_k = R\cos(\theta_k) \qquad \theta_k = \frac{2\pi k}{N}, \; k = 0..N-1$$
$$Y_k = R\sin(\theta_k)$$

Figure 5: *Polar representation of a parametric form*

$$\cos(\theta+\Delta\theta) = \cos\theta\cos\Delta\theta - \sin\theta\sin\Delta\theta$$
$$\sin(\theta+\Delta\theta) = \cos\theta\sin\Delta\theta + \sin\theta\cos\Delta\theta$$

Figure 6: *The trigonometric identity for the cosine of an angle θ and some $\Delta\theta$*

$$\begin{aligned} X_{k+1} &= R\cos(\theta_k+\Delta\theta) \\ &= R\cos\theta_k\cos\Delta\theta - R\sin\theta_k\sin\Delta\theta \\ &= X_k\cos\Delta\theta - Y_k\sin\Delta\theta \\ Y_{k+1} &= R\sin(\theta_k+\Delta\theta) \\ &= R\cos\theta_k\sin\Delta\theta + R\sin\theta_k\cos\Delta\theta \\ &= X_k\sin\Delta\theta + Y_k\cos\Delta\theta \end{aligned}$$

Figure 7: *This equation will generate the points on the circumference of a circle when centered on the origin*

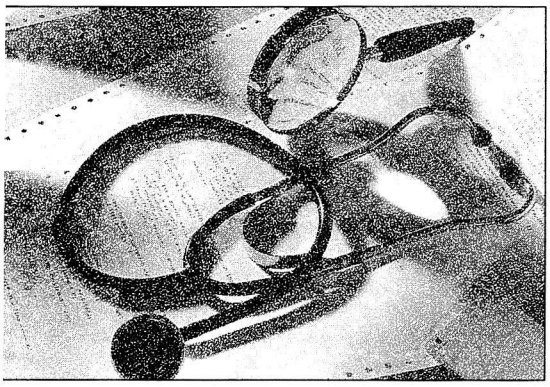
$$X_{k+1} = a + (X_k-a)\cos\theta\Delta - (Y_k-b)\sin\Delta\theta$$
$$Y_{k+1} = b + (X_k-a)\sin\Delta\theta + (Y_k-b)\cos\Delta\theta$$

Figure 8: *This equation will generate the points on the circumference of a circle when the circle is centered on point (a,b)*

My answer to the problem of circle generation begins with the obvious mathematical solution. From there, creative exploration of the solution space starts with trigonometric representations that are not well-suited for computers. However, the application of an identity leaves us with a solution using operations that are an intrinsic part of any computational unit. I've taken this as far as I can go and look forward to your improvements.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 14.**



Examining Zortech C++ 2.0

Putting C++ to the challenge of fractal geometry . . . and more!

Scott Robert Ladd

In mid-1988, Zortech released its native code C++ compiler for the IBM PC and compatibles. Zortech C++, Version 1.0, was based on the prevailing standard of the time — AT&T's C++ 1.x. As a bonus, Zortech C++ was one of the first true source-to-object code C++ compilers in existence. This first version was a very good package, and it boosted the use of C++ on MS-DOS computers. It was with this compiler that I began seriously studying and using C++.

Recently, Zortech released Version 2.0 of its C++ compiler. It was a gigantic step forward, being the first implementation of the AT&T 2.0 version of C++ for MS-DOS. At the same time, Zortech released the first source-level debugger for an MS-DOS C++, improved the programming tools, and enhanced the documentation significantly.

I've been using this compiler in my work (book and article writing) as well as my research into fractal geometry and chaos theory. This article discusses how the product has performed in my C++ projects. Along the way, I'll cover the workings of the compiler and its tools, and the way in which it implements the C++ language.

Scott is a free-lance writer and software developer; he has over 15 years experience in a variety of programming languages. Scott can be reached either at 705 W. Virginia, Gunnison, CO 81230, or via MCI Mail 369-4376. Scott also operates a BBS devoted to computer programming, outdoor activities, and science, at 303-641-5125.

Fractal Geometry

Chaos science, of which fractal geometry is a part, combines mathematics and physics to describe systems which are not understandable by traditional methods. Fractal geometry differs from regular, or Euclidian, geometry in many ways. Euclidian shapes, such as squares and spheres, can be generated and measured using simple formulas. Fractal shapes, however, are created from iterated (and often recursive) algorithms. This is why computers are an integral part of fractal geometry research. In fact, most fractal shapes were not discovered until computers became available for mathematical research.

One of my C++ projects involved writing a program to display a set of fractals similar to the Mandelbrot set. Using a variation of the same recursive algorithm, it is possible to find shapes in the complex plane that closely resemble single-celled life forms. These so-called "biomorphs" can be used to show the dynamics of complex numbers as well as the application of fractal images to modeling microscopic creatures.

In order to develop a C++ program for generating pictures of fractal biomorphs, I needed to have a complex number library. Usually, programs of this type are written in Fortran, which directly supports complex numbers. Fortran, in fact, is very efficient in handling complex number arithmetic. Is it possible to create an efficient complex number class with Zortech C++ 2.0?

The result is shown in Listings One and Two. Listing One (COMPLEX.HPP, page 112) shows the header files for the class Complex, and Listing Two (COMPLEX.CPP, page 114) is the implementation of the non inline methods. To make the class efficient, all methods are defined as inline in Listing One, except those methods that do division, power calculations, and stream I/O. The latter are defined in Listing Two.

Inline methods make complex object calculations quicker than if those methods were actual functions. However, my preliminary results showed that the objects were still 50 percent slower than the equivalent Fortran program. The standard compile options make floating-point calculations use a combined coprocessor/emulator library, which, according to my tests, is 10 to 20 percent slower than equivalent floating-point libraries provided with Microsoft's and Borland's C compilers.

I solved that problem by using the *-f* option when compiling with Zortech C++; that option tells the compiler to generate inline numeric coprocessor instructions. This is both faster and smaller than the standard method of making calls to floating-point emulator routines (which do use the coprocessor, if present). The disadvantage is that a program compiled with *-f* will run only on a computer that contains or emulates a math coprocessor.

This is a good place to point out that Zortech C++ is a globally-optimizing compiler. A section in the reference manual describes how the optional-optimizer pass works. The compiler has three passes. The first pass generates intermediate code, and the third pass converts intermediate code to na-

(continued from page 64)
tive-code form. The optimizer is an optional second pass, which optimizes the intermediate code. This approach is terrific. During development, you can leave out the optimizer pass to get fast compile times. Then, on your final compile, you run the optimizer to generate fast, tight code. My experiments have shown that Zortech C++ 2.0 is competitive with other optimizing compilers such as Microsoft and Watcom.

Integrated Environment

Zortech C++ 2.0 comes with a set of integrated tools. You can edit, compile, and debug your programs in a single environment by using the Zortech Editor (ZED) to control both the compiler and debugger. While this system is not as smooth or quick as the environment provided by other compilers, it offers the advantage of more powerful tools. In a completely integrated environment, such as those that come with Microsoft QuickC and Borland Turbo C, all of the environment's applications are built into one executable file. Thus, you have an editor, debugger, and compiler all in one large .EXE file. This uses a lot of memory and puts restrictions on the power of the individual applications. Many professional programmers disdain integrated environments because of the large memory requirements and limited capabilities of the built-in tools.

This debugger has an amazing number of features, including the ability to provide a profile of program execution

Zortech's approach exchanges some speed and flashiness in order to integrate full-featured applications development tools. The ZED is much improved over its predecessor, and supports multiple file buffers, simple macros, and numerous other features expected of an editor by professional programmers. Instead of building a compiler into the editor, ZED simply calls the command-line compiler and traps the error messages. This isn't as fast or slick as a Turbo Pascal, but it does provide you with the power of a real programmer's editor at a minimal cost.

Debugging

The Zortech debugger (ZDB) is also much improved over the previous version. The interface is easier to use, the mouse support works better than you would expect, and C++ is fully supported. In fact, this is the first source-level C++ debugger for MS-DOS. It manages to handle the infamous C++ "name mangling" (a process whereby C++ internally changes the names of functions and objects to differentiate among overloaded names). ZDB allows you to examine and manipulate identifiers, using the same names you used in your program.

ZDB is a capable debugger; it uses a set of overlapping windows to show everything from source code to registers to local and global variables and constants. One item I didn't like about the debugger was its tendency to "forget" things. If I set up the local's window to expand certain variables, those expansions would be forgotten by ZDB if I stepped into a function call and back.

Another debugger problem occurred when I was debugging the *Complex* class. Because inline methods actually do not exist as functions, there is no way to trace them in the debugger. That's fine and logical. To get around this problem, Zortech provided a compiler switch (-c), which explicitly turns all inline methods into actual functions. This would allow the debugging of these methods. Alas, while the compiler correctly generates the inline methods as functions, it fails to create debugging information for those functions. Thus, while the functions exist, the debugger had no information to trace into them. This is a minor bug in the compiler, not the debugger.

This debugger has an amazing number of features, including the ability to provide a profile of program execution. During months of use both the beta and release versions of the debugger were able to handle everything not mentioned before. An additional bonus is that ZDB is CodeView compatible, and it can debug C and assembly language programs alongside C++.

Is It "Real" C++?

When building the *Complex* class, I realized the need to have input and output routines for *Complex* objects. The obvious approach (and the one I used) is to create functions that use the << and >> operators and the *streams* class. Zortech supports an enhanced version of the original *streams* classes as defined in Stroustrup's book, *The*

(continued on page 69)

(continued from page 66)
C++ Programming Language. AT&T has since enhanced this library itself, but Zortech does not yet support the new AT&T additions (called "iostreams"), saying that AT&T has not made a complete specification available.

The Biomorph program shown in Listing Three, page 119, required a graphics module. Rather than use the Flash Graphics library included with Zortech C++, I linked in the low-level routines from my own C and C++ graphics library. Because of space considerations, the graphics library source code (*zg_lwlvl.h* and *zg_lwlvl.c*) is not included with this article. However, the code is available through the *DDJ* Forum of CompuServe, through the *DDJ* Listing Service, through my BBS, or directly from *DDJ* (see information at the end of this article).

To link these C-language functions with a C++ program required the use of type-safe linkage, one of the more important additions AT&T made to C++ 2.0. Type-safe linkage allows object modules from other languages to be linked with C++ safely. By declaring the native language of a module, the C++ linker knows how names are to be resolved. This lets C++ do whatever it wants with the names in C++ modules, while allowing other languages to do their own thing. In addition to the C and C++ types of linkage, Zortech C++ supports Pascal-style linkage. This is a big plus for those people writing C++ programs for OS/2 and Microsoft Windows.

Did I say Windows? Yes, indeed, Zortech C++ supports Microsoft Windows programming. Zortech does not provide a library of classes or interface routines related to Windows programming. However, the compiler is capable of being used with the Microsoft Windows Software Development Kit (SDK), and documentation is provided that explains how to use the Zortech compiler with the SDK. Because Microsoft C is not too strict about prototypes and other things, from an ANSI standpoint most Windows programs are sloppily written. Zortech is stricter, and you may have to make minor changes to get Windows programs to compile. However, I was able to take several Windows applications, including some from the SDK, and make them compile with Zortech C and C++. While the OS/2 compiler is an extra cost option, the addition of Microsoft Windows compatibility to the MS-DOS compiler will provide the first true C++ compiler for Windows developers.

AT&T Compatibility

Zortech C++ 2.0 is close to the AT&T standard. Zortech does not support pointers to members, the obscure *asm( )* pseudo-function, or fine-grained overload resolution. All of this is documented in the READ.ME file, along with a list of known bugs in the compiler. If only other companies were so honest; most manufacturers seem to make an art of trying to hide the bugs in their products. All compilers have bugs; Zortech is the only one willing to admit to them so they don't sneak up on you. As it is, the bugs in Zortech C++ are relatively trivial, and workarounds are provided when possible.

Zortech C++ does support multiple inheritance, virtual and abstract base classes, anonymous unions, scope resolution, and just about everything else you'd care to throw at it from the *AT&T 2.0 Reference.* I've put some pretty messy code through the Zortech compiler, and it hasn't even blinked.

It is interesting to note that *cfront*, AT&T's own C++, is not entirely compatible with the language as described in the *AT&T C++ 2.0 Reference Man-*

(continued from page 69)
ual. For instance, *cfront* does not support the volatile keyword, which Zortech does. While the Zortech product is not a perfect implementation of the language described in the AT&T standard, it's close. Zortech people have said they will be adding the missing 2.0 features in a subsequent release.

The Goodies

For those of us who have used the Zortech 1.x product, documentation has always been a sore point. To put it bluntly, the documentation with Zortech C++ 1.0 was terrible. That isn't true of the documents that come with Zortech C++ 2.0: These manuals, aside from minor spelling errors, are simply great. *The C++ Compiler Reference* is well-organized and clearly written. It contains a good, if short, C++ tutorial, along with lots of "extra" information on subjects such as the global optimizer and the proper use of C++. *The C++ Function Reference*, *C++ Tools*, and *C++ Debugger* manuals are complete and thorough, including many examples.

In the past, Zortech C++ has been criticized for not being as "fancy" as other compilers. Borland and Microsoft include a number of functions in their C libraries that Zortech did not support. That complaint can now be dropped; Zortech has added a number of functions to its library. Not only have a number of functions been added to increase compatibility with the Microsoft C library, there are new functions to handle expanded memory and TSRs. Library functions carried over from the previous version support graphics, direct-video displays, mouse routines, interrupt trapping, and the PC speaker.

My experiments have shown that Zortech C++ 2.0 is competitive with other optimizing compilers such as Microsoft and Watcom

The compiler has been extended to support a new pointer type, known as a "handle." A handle pointer can be used to access virtual or expanded memory, a major plus for memory-hungry applications. If you don't want to use a language extension, a library of expanded memory functions is also included.

The C++ Tools package is included with the Developer's Edition of the compiler, and is also available separately. Unlike Zortech's, most C++ products fail to provide a significant library of classes. C++ Tools has classes for various kinds of lists, dynamic and virtual arrays, binary trees, hashed tables, BCD math, time and date handling, interrupt handlers, string and text editing, event queues, windows, and money. While some of the classes are not implemented as well as they could be, this is an excellent starting point for those people who do not want to spend time developing their own basic classes.

A "disk 13 of 12" contains a number of other goodies. These include a C++ overlay for the graphics library, an example C++ application, and a library of interesting C routines. While the C++ graphics library is weak, the Doodle program is a good example of C++ program design.

(continued from page 71)

Back to the Biomorphs

The Biomorph program works great. Zortech C++ compiles the program so that it runs at the same speed as the Fortran version. If you have any interest at all in the new science of chaos, or just like to generate interesting pictures on your PC, this program is a great opportunity to experiment. One word of warning: The program performs millions of floating-point operations to generate an image. A PC without a math coprocessor will run Biomorph very slowly; the program is totally unusable on an 8088/86-based PC. It works well on my 20MHz 80386 with 16-bit VGA. Take heart, though — the real researchers on this subject use supercomputers to generate this stuff.

Summing Up

The Zortech C++ compiler has become an intimate part of the work I do. From articles and books on C++ to personal research projects and consulting work, Zortech C++ has had more mileage put on it in the last year than any other compiler in the house. The new 2.0 compiler has improved the product from good to excellent. While it's not perfect, Zortech C++ is one of the best MS-DOS products I've had the luck to use. If you want to work with C++ 2.0 on an MS-DOS computer, I can highly recommend the Zortech 2.0 release.

Currently, Version 2.0 of the Zortech C++ compiler (by itself) costs $200, and the Developer's Edition (including library source code, a C++ class library, and the debugger) costs $450. Considering what this package offers, I'd opt for the more complete Developer's Edition.

Product Information

Zortech C++ V2.0
Zortech, Inc.
1165 Massachusetts Ave.
Arlington, MA 02174
800-848-8408
Developer's Edition (includes compiler, debugger, tools, and library source) $450
C++ compiler $199.95
C++ debugger $149.95
C++ tools $149.95
OS/2 compiler upgrade $149.95

Bibliography

Stroustrup, B. *The C++ Programming Language*, Addison-Wesley, Reading, Mass.: 1985.

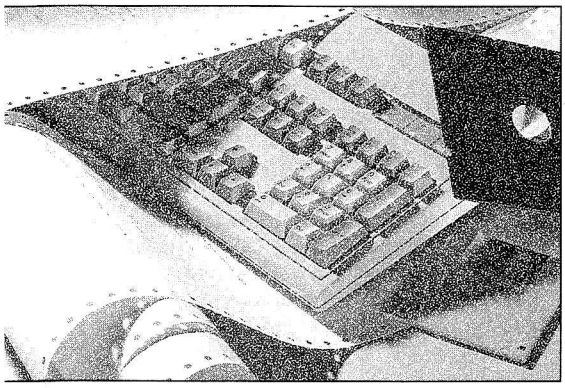
Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 112.)

Vote for your favorite feature/article.
Circle Reader Service **No. 7.**



Stalking GP Faults: Part I

Catching general protection faults when using protected-mode tools

Andrew Schulman

"All protection violations that do not cause another exception cause a general protection exception."
— Intel 80386 Programmer's Reference Manual

" 'Good' exceptions are the ones you expect to occur."
— Edmund Strauss, *Inside the 80286*

In porting applications to the Protected Virtual Address Mode (protected mode for short) of Intel's 286 and 386 processors, many developers have become familiar with the General Protection (GP) fault.

The code in Example 1 causes a GP fault in protected mode. When the Intel processor detects a protection violation (writing into a code segment, peeking past the end of a segment, trying to execute data, or using an illegal segment selector), it raises an exception. A protected-mode operating environment such as OS/2 or a DOS extender catches this exception and, almost always, terminates the offending application.

In contrast to real mode, which allows a program such as this to execute, protected mode makes it possible for an operating system to halt buggy applications. Whereas the real-mode version of this program behaves nondeterministically (or would, if the random number generator were initialized), the protected-mode version behaves the same every time. This makes protected mode terrific for software development.

But, what if a protected-mode environment catches a GP fault from the program in Example 2? That would not be an application problem, but a problem in user input. This program is an interpreter of sorts, and if it GP faults, it's because of an error by the user, not by the application itself.

Now isn't this rather silly program responsible for verifying its input? Perhaps, but these two lines of code represent a large class of programs that allow address manipulation by the user, and in many such extensible systems, it will not be possible to completely verify "user code." It's often better to let the Intel processor handle the verification; your application is then responsible for catching the GP fault signals that the CPU sends.

The GP fault is just an interrupt: INT 0D, in fact. Normally, the operating system installs an interrupt handler for GP faults; this handler halts buggy programs. Clearly, this is not the proper response when the GP fault lies in the user's input rather than your code.

In these cases — when it's not your fault — your application needs to install its own GP fault handler. Depending on the system, catching GP faults may be no more difficult than catching ^C signals, or may be as complicated as running oneself under a pseudo-debug process.

Catching and recovering from GP faults is important to a far wider class of applications than one might guess. Aside from the programmer's interpreters and debuggers, more and more applications include some form of embedded language that allows the user (or, more likely, a consultant) to customize the application. When ported to protected mode, these applications will have to recover gracefully from GP faults. This means building software that is fault-tolerant.

This two-part article discusses catching GP faults in several different 286 and 386 protected-mode DOS extenders and in OS/2. Part I explains why you would want your protected-mode programs to catch GP faults, and then shows how to do so using a 286-based DOS extender. Part II, scheduled to run in the next issue of *DDJ*, shows how to catch GP faults in Phar Lap's 386-based DOS extender, and in IBM and Microsoft's 16-bit OS/2 operating system. Catching GP faults is more difficult in OS/2 than in the other protected-mode environments, and requires use

```
main()
{
    int far *fp = (int far *) main;
    *fp = rand();
    main();
}
```

Example 1: *This code will cause a GP fault in protected mode*

```
main(int argc, char *argv[])
{
    int far *fp = (int far *) atol(argv[1]);
    *fp = atoi(argv[2]);
}
```

Example 2: *If this program GP faults, it's because of an error by the user, not by the application itself*

Andrew is a software engineer in Cambridge, Mass., where he is writing a network CD-ROM server. Andrew can be reached at 32 Andrew (really!) St., Cambridge, MA 02139.

of the interesting *DosPTrace( )* debugging function.

To a programmer familiar with the ANSI C standard library or with the Unix operating system, all of this will at first seem to be much ado about nothing, because the solution is "obviously" to install a SIGSEGV (segmentation violation) handler with *signal( )*. But none of these environments turns an INT 0D into a SIGSEGV signal. For portability with Unix, we could write a SIGSEGV signal handler, and then make our INT 0D interrupt handler raise (SIGSEGV), but that still leaves the question of how to catch INT 0D in the first place.

Because this article focuses on writing code in C to catch GP faults, it also discusses general issues involved with writing interrupt handlers in C, and with the *setjmp( )* and *signal( )* facilities. And, while focusing on one seemingly obscure aspect of 286/386 programming, this article also looks at a lot of different protected-mode tools.

The Protected-Mode PEEK/POKE Problem

"A GP fault is evidence that the program's logic is incorrect, and therefore it cannot be expected to fix itself or trusted to notify the user of its ill health."

Gordon Letwin
Inside OS/2, 1988

Similar statements can be found in the documentation for almost any system based on protected mode, but Letwin is wrong.

How could it be otherwise? Are there any real programs for which a GP fault is not indicative of the program's ill health? How can a program violate the protection model and still deserve to continue executing?

Take the example of Digitalk's superb Smalltalk/V286, which runs in protected mode. Normally Smalltalk/V286 is quite resilient to user errors. But type in this line of code: *Dos new peekFrom: 0 @ 0.* Select it and evaluate it by choosing "Show It" from the menu. (*0 @ 0* is a point in two-dimensional memory space.) The result is that Smalltalk bombs back out to DOS with the message:

Exception 0D: General protection fault
Error code/selector: 0000

All we did was ask to peek at memory location 0000:0000. We weren't even trying to change it! How could peeking at it cause Smalltalk to crash? Still crazy after all those years of real-mode programming, we expected that 0000:0000 would point to the address of the divide-by-zero interrupt vector. In fact, in protected mode dereferencing the address 0000:0000 is guaranteed to be illegal.

Okay, so we're not allowed to peek at 0:0 in protected mode. But the bug was in our code, not in the Smalltalk interpreter. The interpreter was shut down because of a bug in our code? This doesn't sound very protected. In this case, the assumption "a program that GP faults should be terminated" was wrong. It's our "user code" that should be squelched, not the Smalltalk interpreter itself.

To see that this problem is not inherent in protected mode, let's execute the same instruction using another great protected-mode product, Laboratory Microsystems's UR/Forth 386, a 32-bit Forth that runs under Phar Lap's 386|DOS-Extender and virtual-memory manager.

When we execute this Forth statement *00c@L* (fetch from 0:0) and then continue on with the Forth interpreter itself responds with the message: *General protection fault!* This is what Smalltalk/V286 should have done. Instead of crashing back out to DOS, the interpreter stays in control.

That this is easier said than done, though, becomes clear if we lastly look at the OS/2 version of UR/Forth. True, this isn't supposed to be identical to UR/F 386, because that's a completely 32-bit application while the OS/2 version remains, alas, 16-bit. But if we evaluate the same line of code: *0 0*

(continued from page 75)
C@L which, again, means "peek at 0000:0000," OS/2 halts the Forth interpreter and displays a GP fault dump as shown in Figure 1.

Now, unless we're writing a 32-bit protected-mode HastyBasic, few of us care about PEEK and POKE as such. But any other form of memory access can be reduced to PEEK and POKE. The question when writing a protected mode interpreter is what do PEEK and POKE mean in the context of protected mode?

The answer is simple: The user must be allowed to execute any command without fear of killing the interpreter. If the Intel processor detects a GP fault, the interpreter should handle it by printing out a message such as "Illegal selector" or "Can't write to code segment" or "Peeking past end of segment," or perhaps simply "General protection fault!" The interpreter should then return to its top-level input loop.

The Real-Mode GP Fault

Actually, this isn't purely a protected-mode problem. There is also a limited form of GP fault in real mode on 286 and 386 machines. Attempt to dereference more than one byte from offset FFFF:

```
int *p = (int *) -1;
printf("%d \ n", *p);
```

This is an excellent way to crash an IBM AT. The CPU generates an INT 0D, but it isn't caught properly. On an AT, the real-mode GP fault (segment overrun) is treated as though it were a hardware interrupt coming in on IRQ5. IRQ5 can be used by anything from LPT2 to the Novell network to an interrupt-driven SCSI board, so exactly what this code will do depends on your setup.

FFFF.C (Listing One, page 120) is a program that generates this real-mode GP fault on 286 and 386 machines. But it also includes a handler for INT 0D. The listing shows how easy it is to write a real-mode interrupt handler in C. The only difficulty is that, while both Turbo C and Microsoft C push all registers to a function defined with the *interrupt* keyword, they differ in the *order* in which the registers are pushed.

```
Session title: UR/Forth
SYS1943: A program caused a protection violation.
TRAP 000D
AX=2092  BX=0000 CX=FFFF DX=3FC2  BP=FFFA
SI=05EA  DI=05E4 DS=00C7 ES=0000  FLG=2206
CS=0227  IP=2093 SS=00C7 SP=FBFC MSW=FFED
CSLIM=FFFE SSLIM=FFFF DSLIM=FFFF ESLIM=****
CSACC=DF SSACC=F3 DSACC=F3 ESACC=**
ERRCD=0000 ERLIM=**** ERACC=**
End the program
```

Figure 1: *A GP fault dump*

FFFF.C uses a *REG_PARAMS* structure in which the fields appear in a different order depending on the compiler used. *REG_PARAMS* is defined in GPFAULT.H (Listing Two, page 120). The interrupt handler expects one of these structures (not a pointer to it!) to be on the stack. Using 286 instructions, the Microsoft C compiler does a PUSHA; Turbo C pushes each register individually. The processor itself pushes FLAGS and CS:IP before calling the interrupt handler. In addition to distinguishing between the Borland and Microsoft compilers, *REG_PARAMS* also handles the extra parameter that is pushed on the stack of protected-mode handlers for INT 08-0D, and the extra FS and GS registers on the 386. GPFAULT.H is *#include*d by most of the programs in this article.

The real-mode GP fault handler is installed using MS-DOS function 25, just like any other interrupt handler.

How does the INT 0D handler know whether it's been invoked because of a GP fault or because of a hardware interrupt on IRQ5? An INT 0D handler can check whether the CS:IP pushed on its stack corresponds to any of the known places in the program where a GP fault is expected or allowed to occur. If it doesn't, the handler can just pass the interrupt along using the MSC 5.1 *chain_intr( )* function.

This program will not work properly on a 386 machine running in Virtual 86 mode, however. If you run Qualitas' 386 ^ Max, for example, it's as if you hadn't installed the interrupt handler at all:

A Privileged operation exception at address 0ED3:003A.
Press any key to restart your computer.

(In fact, pressing any key has no effect, and you have to reach around the back for the power switch.)

Running FFFF.EXE under Windows/386 doesn't hang your machine like 386 ^ Max, but it does terminate the application, and the message box it displays sounds ominous for a program that simply wants to peek at two bytes starting at offset FFFF (on an 8088, memory would wrap around):

This application has violated system integrity and will be terminated. This may result in the system becoming unstable. It is strongly recommended that you close all applications, exit Windows, and then reboot your machine.

Our exception handler is ignored in

Virtual 86 mode because exceptions in V86 mode are vectored through the protected-mode interrupt descriptor table (IDT), not through the 8086-style interrupt vector table at memory location zero. In fact, it is important that Virtual 86 control programs keep control over INT 0D, because GP fault is precisely the mechanism that programs such as Windows/386 use to virtualize I/O. It would be nice, though, if V86 control programs would not scare us with dire warnings of hellfire, brimstone, and data loss.

Protected-Mode MS-DOS

Stalking the real-mode GP fault is a good introduction to handling GP faults in programs running under a DOS extender. DOS extenders run in protected mode, but switch into real mode to use the INT 21 facilities provided by MS-DOS, to use other INT facilities (for example, the mouse or NetBIOS), or to call functions in real mode (for example, some graphics libraries).

Installing a GP fault handler in a DOS extender is similar to using the MS-DOS *Set Vector* function, and most DOS extenders use the same INT 21 function *25h* interface as real-mode MS-DOS. But, as in V86 mode, *Set Vector* for protected mode manipulates the protected-mode IDT rather than the low-memory interrupt vector table. Also, *Set Vector* in protected mode must be able to install interrupt handlers located in high memory (above one Mbyte). *Set Vector* for a 32-bit environment such as Phar Lap's 386|DOS-Extender must be able to use 32-bit offsets (which, added to 16-bit segment selectors, make for 48-bit *far* addresses). And, finally, because these environments switch back into real mode, DOS extenders also must provide a way to install real-mode interrupt handlers.

GPFAULT.C (Listing Three, page 120) is a program that runs under two 286-based DOS extenders: DOS/16M from Rational Systems, and OS/286 from Eclipse Computing (formerly AI Architects). The phrase 286-based means that these DOS extenders can run on any IBM-compatible equipped with an Intel 80286 *and up*. Developers will presumably be using a 386, but aren't frozen out of the much larger AT market. This 16-bit program can be compiled with Microsoft C 5.1, and then run through a post-processor (DOS/16M MAKEPM or OS/286 EXPRESS). The resulting program, GPFAULT.EXP, can then be spliced with a protected-mode loader to form GPFAULT.EXE.

The GPFAULT program is a mock interpreter that allows the user to try to poke at arbitrary locations in memory, as shown in Figure 2, the session with the DOS/16M version.

It is interesting that while poking at 1234:5678, 0088:00CB, and 0000:0000, all caused GP faults that happened in different ways. In fact, we can see why this is called the *general* protection fault: It is a catchall for everything that doesn't fit into some other exception.

In the example session, 1234:5678 was a completely bogus pointer, and the processor refused even to load 1234 into a segment register (note that ES still contains the selector 00A0). We didn't get anywhere near trying to poke this address: There was no such address in this session. In contrast to real mode, in which every segment:offset combo points *somewhere* (in this sense, real mode is magic), memory in protected mode is a sparse matrix: Most addresses you can form don't point anywhere.

Because the processor refuses to load the number, the GP fault handler can't find it in the registers and, consequently, has no way of knowing what caused the fault, unless the processor also gives it some additional information.

Fortunately, the Intel processor provides this information by passing an extra parameter on the stack of a GP fault handler. When GPFAULT.H is included in a protected-mode program, the *REG_PARAMS* structure contains a field for this error code. In a C interrupt handler, this extra parameter goes between the CS:IP and FLAGS pushed by the processor and the PUSHA supplied by the C compiler.

The handler could use the Intel LAR and LSL instructions to figure out why a GP fault took place (did we try to write into code, did the offset overrun the segment limit, and so on), so that it could provide the user with a more informative message or perhaps *correct* the error.

Trying to poke 0088:00C5 did not produce an error code (the processor just pushes a 0 on the handler's stack). This is because 0088 was a valid segment selector; in fact, it was GPFAULT .EXP's code segment. This means that loading it into ES was a legal operation (as we can see from the register dump), but trying to poke it caused a GP fault.

This explains why the CS:IP where the fault took place was a few bytes further down from where we tried poking the totally bogus address. In GPFAULT.C, the crucial line of code

```
*fp = data;
```

becomes three instructions:

```
les bx, dword ptr _fp
mov ax, word ptr _data
mov word ptr es:[bx], ax
```

Loading a bogus segment selector faults on the first instruction, whereas doing something illegal with a genuine segment selector faults on the third.

Trying to poke memory location zero is a special case. While segment selec-

```
C:\DOS16M>gpfault
DOS/16M Protected Mode Run-Time  Version 3.25
Copyright (C) 1987,1988,1989 by Rational Systems, Inc.
'Q' to quit, '!' to reinstall default GP fault handler
00A0:04C6 is a legal address to poke
00A0:04C4 is not a legal address to poke
$ 1234:5678 666
Protection violation at 0088:00C5!
Error code 1234
<ES 00A0> <DS 00A0> <DI 1AC0> <SI0082>
<AX 0015> <BX 0BF8> <CX 0015> <DX 0000>
$ 00A0:04C4 666
poked 00A0:04C4 with 666
$ 00A0:04C2 1
poked 00A0:04C2 with 1
$ 0088:00C5 666
Protection violation at 0088:00CB!
<ES 0088> <DS 00A0> <DI 1AC0> <SI 0082>
<AX 029A> <BX 00CB> <CX 0015> <DX 0000>
$ 0:0 0
Protection violation at 0088:00CB!
<ES 0000> <DS 00A0> <DI 1AC0> <SI 0082>
<AX 0000> <BX 0000> <CX 0015> <DX 0000> $ !
$!
$ 0:0 0
DOS/16M: Unexpected Interrupt=000D at 0088:00CB
code=0000 ss=00A0 ds=00A0 es=0000
ax=0000 bx=0000 cx=0015 dx=0000 sp=1982 bp=1A92 si=0082 di=1AC0
C:\DOS16M>
```

Figure 2: *A GPFAULT.EXP session*

tor 0 is bogus in the sense that it is guaranteed not to correspond to an actual segment descriptor, loading 0 into a segment register is not illegal: It's using any resulting pointer that's illegal. In this way, the Intel processor in protected mode lends support to high-level languages that treat memory location zero as the NIL pointer.

While on the subject of 0:0, remember that even trying to peek at this location caused Smalltalk/V286 and UR/Forth OS/2 to blow up. Here we tried to poke it, but because we installed an INT 0D handler, the mock-interpreter stayed in control. At the end of the session, I told GPFAULT to reinstall DOS/16M's default GP fault handler, and then did another POKE 0:0 0. The result this time was that DOS/16M spewed out some diagnostics and the interpreter halted. This is the sort of behavior we've avoided by installing our own handler.

The IRQ5 Problem

Microsoft C 5.1 provides the function *dos_setvect( )* to install an interrupt handler. Turbo C provides *setvect( )*. These functions merely call INT 21 function 25. Can these functions be used from a protected-mode DOS extender to install a GP fault handler?

All DOS extenders provide an INT 21 interface for protected-mode programs (that, in fact, is a pretty good definition of a DOS extender), and that includes INT 21 function 25. This means that a lot of even low-level code, including a call to *_dos_setvect( )*, works transparently under a DOS extender.

However, the GP fault is a little unusual. INT 0D lies in the range of Intel processor exceptions that are also IBM AT-compatible hardware interrupts. On the 286 and above, Intel reserved INT 0D and other slots for processor exceptions like the GP fault, but IBM wasn't thinking about protected mode at the time and overloaded INT 08 through 0F for use by the AT hardware interrupts IRQ0 through IRQ7. Thus, INT 0D is multiplexed between the GP fault and IRQ5 (which, in turn, may be used by LPT2 or by an Ethernet board or by an interrupt-driven SCSI board or by . . .).

The 386-based DOS extender from Phar Lap and Quarterdeck's DESQview 386 both solve this problem by reconfiguring the 8259 programmable interrupt controller (PIC) so that IRQ0 through IRQ7 are relocated to INT 78 through 7F (see Ray Duncan, "Power Programming," *PC Magazine*, October 17, 1989).

But 286-based DOS extenders happen not to reprogram the 8259 PIC. So, when you install an INT 0D handler with INT 21 function 25, are you installing a protected-mode GP fault handler or a real-mode IRQ5 handler? The 286-based DOS extenders try to "Do the Right Thing" (to trivialize the title of a recent movie). And alternatives to INT 21 function 25 are provided when the extender's default behavior isn't what you need.

Returning to GPFAULT.C in Listing Three, for DOS/16M, I used the functions *D16pmInstall( )* and *D16pmGetVector( )*. The "pm" in the function names stands for protected mode. These functions, along with many other C interface routines for protected-mode programming, appear in the file DOS-16LIB.C that comes with DOS/16M and with Rational Systems's protected-mode development environment, Instant-C.

The Limits of Protection

In addition to illustrating how to catch the GP fault, Listing Three also shows its limitations. Having committed one GP fault and seeing the register dump, the user now knows the selector number for DS, and an error prone or malicious user can freely poke at the program's internal data.

Doing so doesn't cause a GP fault. Instead, the more places the user clobbers, the more erratically the program behaves. Generally, the interpreter's

strings are clobbered first. Because they take up more room in memory than other variables, there's a good chance that random pokes will first hit strings. Soon, crucial program data is poked, and the program itself causes a GP fault. This is somewhat like the game Core Wars (which first appeared in the "Computer Recreations" column of *Scientific American*), in which two assembly-language programs try to make each other execute an illegal opcode.

A GP fault handler must be able to distinguish its own "internal" GP faults from those caused directly by user code. GPFAULT.C keeps a "whereami" flag. The GP fault handler checks whether *(whereami == IN_USER_CODE)* which indicates whether INT 0D occurred in any known places in the code where a GP fault is expected or allowed to occur. If it didn't, then the INT 0D was caused by an internal error (or by a real mode IRQ5, though in that case the processor won't have pushed the additional error code, and our interrupt handler expecting a *REG_PARAMS* on the stack will view CS and IP as garbage). If the fault took place while directly executing code on behalf of the user, then we print out some diagnostics and *longjmp( )* back to the interpreter's main input loop. *longjmp( )* is a non-local goto. The *jmp_buf* holds the state of the program's registers at the time we called *setjmp( )*, and calling *longjmp( )* restores this state. The program's thread of execution resurfaces as if it had just "returned" from *setjmp( )*.

While this article focuses on writing code in C to catch GP faults, it also discusses general issues involved with writing interrupt handlers in C, and with the setjmp() *and* signal() *facilities*

Signal handlers that execute *longjmp( )* are notoriously unreliable. So you might have some doubts about taking a *longjmp( )* from within an interrupt handler. For example, what about all the junk that was pushed on the stack at entry to the interrupt handler? If we *longjmp( )* out of the interrupt handler, we never execute the IRET instruction and, it seems, we never clean up the stack. The *jmp_buf*, however, includes the stack pointer from when *setjmp( )* was first called, so *longjmp( )*, by loading this saved stack pointer into SP, will effectively cut back the stack.

There's still a problem with an interrupt handler that doesn't return: It can leave the program in an incomplete state. By itself, *longjmp( )* is insufficient for true exception handling. In a real program, the *jmp_buf* should probably be enclosed within a larger application-defined structure that includes other fields necessary for cleanup (such as open file handles, data base indexes, and so on). For a good discussion of *longjmp( )*'s problems as an exception-handling mechanism, see William M. Miller's paper, "Exception Handling without Language Extensions," *Usenix Proceedings C++ Conference*, Denver, Colo., 17 – 21 October, 1988.

What does the fault handler do if the flag indicates the program wasn't in

user code? The GP fault must then have been caused by our own code (perhaps because the user tromped on some of our data, including possibly the *whereami* flag itself). In this program, I print out a message and then use *chain_intr( )* to call the default GP fault handler, which terminates the application. In a real program, you might want to try to give users a chance to save their work, perhaps by disabling all menu items except "File Save. . ." and displaying a "The End Is Near" message.

Because the *whereami* flag itself is so easy to tromp on, it holds "magic" numbers rather than simple enum values. And because the *jmp_buf* top level is so crucial, the program maintains two of them. In a real program, you might want to make such essentials read only while executing user code. Even better, you might want to try to run user code using a different local descriptor table (LDT) from the one you use for the interpreter itself.

Any protection scheme has similar limitations. Who protects the protector? Another protector, of course, almost (but not quite) as vulnerable. Ultimately, the protection problem is the same as the halting problem.

This concludes Part I of the epic saga, "Stalking GP Faults." Tune in again next month, when our hero gets caught inside 32-bit code running under a 386-based DOS extender, and inside 16-bit code running under the OS/2 operating system. Until then, remember: To commit a GP fault is human, but to catch it and recover, divine.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 120.)

Vote for your favorite feature/article.
Circle Reader Service **No. 8.**

Listing One *(Text begins on page 16.)*

```
        .model  small, c
        .286    ; This directive can be used to optimize procedure entry
                ; but not much else.  I avoided all non-8088 commands
        comment \
                        EGA Sprite Drivers for C
                Copyright (c) February 1989, Ryu Consulting, Inc.
                        (916) 722 - 1939 anytime
                        Written by Rahner James, CS
This is a full functioning sprite driver for EGA graphics adaptors. The
sprite are given to the routines as a linked list of sprite structures.  All
the function are re-entrant and can be part of a multi-tasking system.  The
sprite structures are intended to reside in far memory.  The sprite can
exist on any pixel boundary. These were intended to be called from some C
program, but probably can be modified for some other language.  This must be
assembled with Microsoft MASM version 5.0 or later since I make use of local
variables, forward/backward jumps and models.  Expect to get some incorrect
size warnings because MASM doesn't seem to recognize its own "byte ptr" and
"word ptr" operators when used with words and dwords.\
; ***************************************************************************
;       EQUATES
; ***************************************************************************
EOI             equ     20h             ; End Of Interrupt signal
EOI_PORT        equ     20h             ; Port to output the EOI
CRT_MODE        equ     49h
EGA_ADDRESS     equ     63h

EGA_PIXELS_WORD equ     16              ; Number of pixels per word
EGA_PIXELS_BYTE equ     8               ; Number of pixels per byte
NUMBER_OF_PLANES equ    4               ; Number of EGA color planes

EGA_RETRACE_STATUS      equ     3dah    ; EGA retrace status register
RETRACE_BIT             equ     1 shl 3 ; Bit set to signal a vertical retrace
SEQUENCE_REG            equ     3c4h    ; Sequencer register
GRAPHICS_12             equ     3ceh    ; Graphics 1 & 2 register
MAP_MASK_REG            equ     2       ; Map mask Indexed register
DATA_ROTATE_REG equ     3       ; Data Rotate Indexed register
DATA_OR         equ     1 shl 4 ; Set to OR data on the EGA
DATA_MOVE               equ     0       ; Write data unmodified onto EGA

BOTTOM_LINE             equ     200     ; Lowest pixel line to allow a sprite
RIGHT_SIDE              equ     640     ; Right-most visual pixel allowed
; ***************************************************************************
;       STRUCTURES
; ***************************************************************************
; All sprites are stored on disk using the same internal format. The first word
; is the width of the sprites in bytes and the second word is the height of the
; sprite in widths.  Each byte represents ; one pixel's worth of information.
; Bit 7 of the byte is the intensity bit, 0=off.  This allows for 128 colors,
; black will be 0 or 80h.  The intensity bit set indicates an opaque black
; surface. All color translations are table driven.

internal_sprite_structure       struc   ; Storage structure used for sprites
int_width       dw      ?               ; Width in bytes for the sprite
int_height      dw      ?               ; Height in widths of the sprite
int_body        db      ?               ; Start of the sprite's body
internal_sprite_structure       ends

ega_sprite_structure    struc ; Internal EGA sprite structure
e_animate_ptr   dw      0,0   ; Far ptr to next sprite struct in animation seq.
e_width dw      ?       ; Width of the sprite in words
e_height        dw      ?     ; Height of sprite in widths
e_body          dw      ?     ; Beginning of body
                              ;.word 0: mask, word 1: sprite
                              ;.throughout the body.  That way you can pull
                              ;.the background up, mask it, OR the sprite,
                              ;.then store the background
                              ; The body is organized into four planes,
                              ;.termed PLANE0 to PLANE3.  Each represents a
                              ;.different color in the EGA spectrum, except
                              ;.PLANE0 which is the intensity bit.
ega_sprite_structure    ends

style_structure struc
style_width     dw      ?               ; Width of each style entry in bytes
style_height    dw      ?               ; Height of each style entry in pixels
style_body      db      ?               ; Start of the style entries
style_structure ends
; ***************************************************************************
;       LOCAL DATA STORAGE for DS
; ***************************************************************************
.data
public          do_page_flip, done_page_flip
        even
do_page_flip    db      0       ; Set to -1 when non-visual page is completed
done_page_flip  db      -1      ; Set to -1 right after page has been swapped
flip_turn       db      4       ; # of interrupts before an EGA page flip

old_irq_mask    db      0       ; Old IRQ mask

EGA_settings    db      2bh,2bh,2bh,2bh,24h,24h,23h,2eh
                db      0,0,0,0,0,24h,23h,2eh,2bh

        even
ega_base_port           dw      ?
default_retrace label   word
v_retrace_reg           db      11h
v_retrace_value db      ?

.code
; ***************************************************************************
;       LOCAL DATA STORAGE for CS
; ***************************************************************************
        even

ega_segment  dw  0a800h ; EGA page memory segment being set up
old_vector   dw  0,0    ; Old IRQ-2 vector (as Checkov would say wecter)

; EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
; EGA sprite related routines below this line
; void  EGA_CONVERT( dest_ptr, source_ptr )
; Converts long storage format sprite into EGA structure sprite
; Given:
;       dest_ptr far -> EGA sprite buffer ready to go
;       source_ptr far -> disk sprite structure to convert
; Returns:
;       EGA sprite buffer set up accordingly

ega_convert proc near uses di si ds, dest:dword, source:dword
local   store_width:word, store_height:word
local   source_width:word
local   plane_size:word
        cld
        lds     si, source              ; DS:SI -> source sprite
        les     di, dest                ; ES:DI -> CGA sprite buffer

        mov     es:[di], di             ; Make the animate ptr point to itself
        mov     es:[di+2], es
        add     di, E_WIDTH

        lodsw                           ; Get width in byte pixels
        mov     source_width, ax
        add     ax, EGA_PIXELS_WORD-1   ; Want to include those border pixels
        mov     cl, 4                   ; Divide by sixteen to convert one byte
        shr     ax, cl                  ;.per pixel to 16 pixels/word for EGA
        mov     bx, ax                  ; Use this as our width count
        stosw                           ; Store as words/pixels
        lodsw                           ; Get height
        mov     store_height, ax
        stosw                           ; Move height straight across
        mul     bx                      ; AX = Body size in mask/sprite entries
        add     ax, ax                  ; AX = body size in words
        add     ax, ax                  ; AX = plane size in bytes
        mov     plane_size, ax          ; Save as our plane index

        mov     ax, ds                  ; Swap DS:SI and ES:DI
        mov     bx, es
        mov     ds, bx
        mov     es, ax
        xchg    di, si

        sub     si, 4                   ; This is to prep for the next INC
next_row:
        mov     cx, source_width        ; Get source row width
next_word:
        add     si, 4                   ; SI -> next word in line
        mov     dx, -1                  ; DX = the destination mask
        xor     ax, ax
        mov     [si], dx                ; Set the mask word
        mov     [si+2], ax              ; Clear the sprite word
        mov     bx, plane_size          ; BX offset to next plane
        mov     [si+bx], dx             ; Set the mask word
        mov     [si+bx+2], ax           ; Clear the sprite word
        add     bx, plane_size          ; BX offset to next plane
        mov     [si+bx], dx             ; Set the mask word
        mov     [si+bx+2], ax           ; Clear the sprite word
        add     bx, plane_size          ; BX offset to next plane
        mov     [si+bx], dx             ; Set the mask word
        mov     [si+bx+2], ax           ; Clear the sprite word

        mov     dx, 1 shl 7             ; Start at MSB which is pixel LSB
next_pixel_byte:
        jc      next_word               ; Only be set by pixel shift below
        mov     al, es:[di]             ; Get the source pixel byte
        inc     di                      ; DI -> next source pixel byte
        or      al, al                  ; See if it's anything at all
        jz      end_pixel_byte          ; Skip all the checks

        xor     [si], dx                ; Reset the mask bit
        test    al, 1 shl 4             ; Check bit 7
        jz      @F                      ; Skip if nothing here
        or      [si+2], dx              ; Place the sprite bit
@@:     mov     bx, plane_size          ; BX -> plane 1 offset
        xor     [si+bx], dx             ; Clear the mask bit
        test    al, 1 shl 7             ; Check bit 6
        jz      @F                      ; Skip if nothing here
        or      [si+bx+2], dx           ; Place the sprite bit
@@:     add     bx, plane_size          ; BX -> plane 2 offset
        xor     [si+bx], dx             ; Clear the mask bit
        test    al, 1 shl 6             ; Check bit 5
        jz      @F                      ; Skip if nothing here
        or      [si+bx+2], dx           ; Place the sprite bit
@@:     add     bx, plane_size          ; BX -> plane 3 offset
        xor     [si+bx], dx             ; Clear the mask bit
        test    al, 1 shl 5             ; Check bit 4
        jz      end_pixel_byte          ; Skip if nothing here
        or      [si+bx+2], dx           ; Place the sprite bit
end_pixel_byte:
        shr     dl, 1                   ; Move pixel bit toward MS pixel bit
        rcr     dh, 1
        loop    next_pixel_byte         ; Loop through the pixel bytes

        dec     store_height            ; One less row
        jnz     next_row

        ret
ega_convert     endp
; EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
; ui EGA_CALCULATE( source_ptr )
; Calculates the amount of storage needed for an unconverted sprite
; Given:
;       source_ptr far -> disk sprite structure to convert ; Returns:
;       AX = number of bytes needed to store the converted sprite

ega_calculate proc near uses si ds, source:dword
        lds     si, source      ; DS:SI -> sprite structure for calculation
```

(continued on page 84)

Listing One *(Listing continued, text begins on page 16.)*

```
        mov     ax, [si]        ; Get the width in bytes
        add     ax, EGA_PIXELS_WORD-1   ; Round up to nearest word
        shr     ax, 4                   ; AX = number of words for stoarge
        add     ax, ax                  ; AX = number of bytes storage
        add     ax, ax                  ; AX = number of globs in one row
        add     ax, ax                  ; AX = number of row/planes
        add     ax, ax
        mul     word ptr [si].int_height ; AX = bytes per row * number of rows
        add     ax, E_BODY              ; AX = body size + header size
        ret
ega_calculate   endp
; EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
; void PUT_SPRITE( ui X, ui Y, ega_sprite_structure far *SPRITE_PTR )
; Maps a sprite list into the display video buffer
; Sprite is assumed to be 32-bits wide
; Given:
;       X = X pixel location of sprite
;       Y = Y pixel location of sprite
;       SPRITE_PTR -> sprite structure to put on the screen
; Returns:
;       The sprite is mapped onto the screen buffer
put_sprite proc uses di si, x:word, y:word, sprite_ptr:dword

local   mask_msw:word                   ; OR'ed with the mask dword
local   mask_lsw:word
local   or_msw:word                     ; AND'ed with the sprite dword
local   or_lsw:word
local   start_di:word
local   start_si:word
local   sprite_plane_size:word
local   number_of_rows:word
local   shift:byte
local   plane_number:byte
        push    ds
        cmp     y, 200                  ; See if below 200
        jc      @F
short_done:
        jmp     done
@@:     cmp     x, 640
        jnc     short_done

        cld
        mov     es, ega_segment
        lds     si, sprite_ptr          ; DS:SI -> sprite to be driven

        mov     ax, 80                  ; Calculate the offset
        mul     y
        mov     cx, x
        mov     di, cx
        and     cl, 7                   ; CL = shift value
        mov     shift, cl
        shr     di, 3
        add     di, ax                  ; DI -> byte offset
        mov     start_di, di

        mov     ax, [si].e_height       ; AX = height of the sprite in rows
        mov     number_of_rows, ax      ; Save for later use
        shl     ax, 3                   ; AX *= 8, to get size of 1 sprite plane
        mov     sprite_plane_size, ax

        mov     ax, number_of_rows      ; Let's see if it goes too low
        add     ax, y
        sub     ax, BOTTOM_LINE
        jbe     @F                      ; Skip if it doesn't
        sub     number_of_rows, ax      ; Update the number of rows

@@:     add     si, E_BODY              ; SI -> start of sprite body
        mov     start_si, si

        cmp     x, 640-32               ; See if we are going to be right
        ja      short_mask              ; Skip if no mask
        cmp     shift, 0
        jnz     shifted

        mov     bl, NUMBER_OF_PLANES
next_plane:
        dec     bl

        mov     dx, GRAPHICS_12         ; Talk to EGA control logic
        mov     al, 4
        mov     ah, bl
        out     dx, ax
        mov     dx, SEQUENCE_REG        ; Set up the ports for writing as well
        mov     ax, 100h + MAP_MASK_REG
        mov     cl, bl
        shl     ah, cl
        out     dx, ax

        mov     cx, number_of_rows
@@:     mov     ax, es:[di]             ; Get the background dword
        mov     dx, es:[di+2]
        and     ax, [si]                ; Do the mask dword
        and     dx, [si+4]
        or      ax, [si+2]              ; Bring on the sprite
        or      dx, [si+6]
        mov     es:[di], ax             ; Replace with new graphic dword
        mov     es:[di+2], dx
        add     si, 8                   ; Next row stuff
        add     di, 80
        loop    @B

        mov     di, start_di
        mov     si, start_si
        add     si, sprite_plane_size
        mov     start_si, si
        or      bl, bl
        jnz     next_plane
        jmp     done

short_mask:
        jmp     masked

shifted:
        mov     plane_number, NUMBER_OF_PLANES-1
        mov     cl, shift
        mov     bh, -1
        shr     bh, cl
next_shift_plane:
        mov     dx, GRAPHICS_12         ; Talk to EGA control logic
        mov     al, 4
        mov     ah, plane_number
        out     dx, ax

        mov     dx, SEQUENCE_REG        ; Set up the ports for writing as well
        mov     ax, 100h + MAP_MASK_REG
        mov     cl, plane_number
        shl     ah, cl
        out     dx, ax

        mov     ch, byte ptr number_of_rows
        mov     cl, shift
@@:     lodsw                           ; Get the sprite mask
        xchg    ah, al                  ; Switch them around
        ror     ax, cl
        mov     bl, ah                  ; Top CL bits are ones to mask 3rd byte
        not     bh                      ; BH = ~BH
        and     bl, bh
        or      ah, bh                  ; Now set the top ones of source byte
        mov     dx, es:[di]             ; Get the first destination word
        xchg    ah, al                  ; Re-order the mask bytes
        and     dx, ax                  ; Mask it
        lodsw                           ; Get the sprite
        xchg    ah, al
        ror     ax, cl
        or      dh, al                  ; OR DH w/ old AH
        mov     al, ah                  ; Save the upper bits
        and     ah, bh                  ; Mask off other bits
        not     bh                      ; BH = BH
        and     al, bh                  ; Cut out the rotunds
        or      dl, al                  ; OR least sig. bytes
        mov     es:[di], dx             ; Save that first word, whew!

        mov     dl, ah                  ; DL = pushed up sprite bits

        lodsw                           ; Get the next sprite mask
        xchg    ah, al                  ; Switch them around
        ror     ax, cl
        mov     dh, ah                  ; DH = MS shifted mask bits
        and     ah, bh                  ; Get rid of shifted bits
        or      ah, bl                  ; OR with shifted mask, previous byte
        or      dh, bh                  ; Add on the mask
        xchg    ah, al                  ; AH:AL back to normal
        and     es:[di+4], dh           ; Easy way to get rid of DH
        mov     bl, dl                  ; BL = previous sprite bits
        mov     dx, es:[di+2]           ; Get the destination word
        and     dx, ax                  ; Mask it
        lodsw                           ; Get the sprite
        xchg    ah, al
        ror     ax, cl
        or      dh, al                  ; OR DH w/ old AH
        mov     al, ah                  ; Save the upper bits
        not     bh                      ; BH = ~BH
        and     ah, bh                  ; Mask off other bits
        not     bh                      ; BH = BH
        and     al, bh                  ; Cut out the rotunds
        or      al, bl
        or      dl, al                  ; OR least sig. bytes
        mov     es:[di+2], dx           ; Save that first word, whew!
        or      es:[di+4], ah

        add     di, 80

        dec     ch
        jnz     @B

        mov     di, start_di
        mov     si, start_si
        add     si, sprite_plane_size
        mov     start_si, si
        sub     plane_number, 1
        jc      @F
        jmp     next_shift_plane
@@:     jmp     done

masked:
        xor     ax, ax                  ; Set up masks and ORs
        mov     mask_lsw, ax
        mov     mask_msw, ax
        dec     ax
        mov     or_lsw, ax
        mov     or_msw, ax

        cmp     x, 640-24               ; See if we have masked it already
        jc      @F                      ; Skip if we have
        mov     mask_msw, 0ff00h
        mov     or_msw, 0ffh

        cmp     x, 640-16
        jc      @F
        mov     mask_msw, -1            ; Make sure nothing gets masked
        mov     or_msw, 0

        cmp     x, 640-8                ; See if that's all
        jc      @F
        mov     mask_lsw, 0ff00h
```

(continued on page 86)

Listing One *(Listing continued, text begins on page 16.)*

```
        mov     or_lsw, 0ffh

@@:     mov     plane_number, NUMBER_OF_PLANES-1
next_mask_plane:
        mov     dx, GRAPHICS_12         ; Talk to EGA control logic
        mov     al, 4
        mov     ah, plane_number
        out     dx, ax

        mov     dx, SEQUENCE_REG        ; Set up the ports for writing as well
        mov     ax, 100h + MAP_MASK_REG
        mov     cl, plane_number
        shl     ah, cl
        out     dx, ax

        mov     cx, number_of_rows
@@:     mov     ax, [si]                ; Get the first mask
        mov     dx, [si+4]              ; Get the second mask
        or      ax, mask_lsw
        or      dx, mask_msw
        and     ax, es:[di]             ; Get the background dword
        and     dx, es:[di+2]
        mov     bx, or_lsw              ; Get the OR lsw
        and     bx, [si+2]              ; Bring on the sprite
        or      ax, bx
        mov     bx, or_msw
        and     bx, [si+6]
        or      dx, bx
        mov     es:[di], ax             ; Replace with new graphic dword
        mov     es:[di+2], dx
        add     si, 8                   ; Next row stuff
        add     di, 80
        loop    @B

        mov     di, start_di
        mov     si, start_si
        add     si, sprite_plane_size
        mov     start_si, si
        sub     plane_number, 1
        jnc     next_mask_plane
        jmp     done

done:   pop     ds
        ret
put_sprite      endp
; EEEE
```

Listing One *(Listing continued, text begins on page 16.)*

```
        mov     dx, 3d4h
        mov     ax, 2b11h
        out     dx, ax
        sti
done:   ret
ega_rip_out endp
; EEEE
```

Listing Two *(Listing continued, text begins on page 16.)*

```
        jnc     pre_next_sprite
        mov     ax, [si].x              ; See if we need to update some things
        cmp     ax, 640                 ; See if beyond right column
        jnc     pre_next_sprite
        cmp     ax, pre_min_x           ; See if pre_x < min_x
        jnc     @F                      ; Skip if not
        mov     pre_min_x, ax           ; Update with new MIN_X

@@:     mov     ax, [si].y              ; See if need to update min_y
        cmp     ax, pre_min_y
        jnc     @F
        mov     pre_min_y, ax

@@:     les     bx, dword ptr [si].sprite_ptr   ; ES:BX -> sprite structure

        mov     ax, es:[bx].sprite_width
        inc     ax
        if      (@Cpu AND 2)
        shl     ax, 4                   ; AX = AX * 16
        else
        rept    4
        add     ax, ax
        endm
        endif
        add     ax, [si].x
        cmp     pre_max_x, ax           ; See if > max_x
        jnc     @F                      ; Jump if not
        mov     pre_max_x, ax           ; Assume we are going to save it

@@:     mov     ax, es:[bx].sprite_height
        add     ax, [si].y
        cmp     pre_max_y, ax           ; See if > max_x
        jnc     @F                      ; Jump if not
        mov     pre_max_y, ax           ; Assume we are going to save it

@@:     push    es                      ; Set up for call to put_sprite()
        push    bx
        push    [si].y
        push    [si].x
        call    put_sprite
        add     sp, 4                   ; Clear the stack
        pop     bx                      ; ES:BX -> sprite pointer
        pop     es
        les     bx, dword ptr es:[bx].animate_ptr
        mov     [si].sprite_ptr, bx
        mov     [si].sprite_ptr+2, es

pre_next_sprite:
        mov     si, [si].next_node      ; SI -> next sprite node in line
        jmp     next_sprite

almost_done:
        mov     ax, pre_max_x
        cmp     ax, 640
        jb      @F
        mov     ax, 639
        mov     pre_max_x, ax
@@:     cmp     pre_min_x, ax
        jb      @F
        dec     ax
        mov     pre_min_x, ax
@@:     mov     ax, pre_max_y
        cmp     ax, 200
        jb      @F
        mov     ax, 199
        mov     pre_max_y, ax
@@:     cmp     pre_min_y, ax
        jb      @F
        mov     pre_min_y, ax

@@:     mov     done_page_flip, 0
        mov     do_page_flip, -1

done:   ret
do_sprite_list endp
; ***************************************************************************
; void CLEAR_SPRITE_LIST( void )
; Zeros out the present sprite list
; Given:
;       nothing
; Returns:
;       FIRST_SPRITE and SPRITE_LIST array are zeroed

clear_sprite_list proc uses di

        cld
        mov     di, offset sprite_list
        mov     ax, ds
        mov     es, ax
        xor     ax, ax
        mov     first_sprite, ax
        mov     cx, (MAX_SPRITES * (size sprite_node))/2
        rep     stosw

        ret
clear_sprite_list endp
; ***************************************************************************
; int INSERT_SPRITE( ui X1,ui Y1, ui D_X,ui D_Y, ui PLUS_X,ui PLUS_Y,
;                    ui THE_DEPTH, sprite_structure far *SPRITE )
; Inserts the first sprite of a sprite circle into the linked list of circles
; Given:
;       X1,Y1 = pixel location of the upper left corner of the sprite
;       D_X,D_Y = pixel location of the destination of the sprite
;       PLUS_X,PLUS_Y = pixels the sprite moves every page flip
;       THE_DEPTH = apparent distance of the sprite from the viewer
;       SPRITE -> sprite to insert
; Returns:
;       If 0, sprite pointer was inserted in the array
;       else !0 if there is no room

insert_sprite proc uses si, x1:word,y1:word, d_x:word,d_y:word,\
                            plus_x:word,plus_y:word, the_depth:word,\
                            sprite:dword

        mov     cx, MAX_SPRITES
        mov     bx, (offset sprite_list) - (size sprite_node)
@@:     add     bx, size sprite_node
        mov     ax, [bx].sprite_ptr     ; See if this has been set yet
        or      ax, [bx].sprite_ptr+2
        loopnz  @B
        jnz     done_bad

        les     ax, sprite              ; ES:AX -> sprite location
        mov     [bx].sprite_ptr, ax     ; Save it
        mov     [bx].sprite_ptr+2, es

        mov     ax, x1                  ; Set up the structure
        mov     [bx].x, ax
        mov     [bx].pre_x, ax
        mov     ax, y1
        mov     [bx].y, ax
        mov     [bx].pre_y, ax
        mov     ax, d_x
        mov     [bx].dest_x, ax
        mov     ax, d_y
        mov     [bx].dest_y, ax
        mov     ax, plus_x
        mov     [bx].adder_x, ax
        mov     ax, plus_y
        mov     [bx].adder_y, ax
        mov     ax, the_depth
        mov     [bx].depth, ax          ; This is used in the following loop
        mov     dx, bx                  ; Save this sprite entry for later

        mov     si, first_sprite        ; SI -> sprite furthest from the viewer
        mov     bx, offset first_sprite ; BX -> previous sprite entry
        mov     cx, MAX_SPRITES
@@:     or      si, si                  ; See if it's a NULL ptr
        jz      @F                      ; Skip out if it is
        cmp     ax, [si].depth          ; See if farther from observer
        jnc     @F
        mov     bx, si                  ; BX = this pointer
        mov     si, [si].next_node      ; SI -> next sprite node in line
        loop    @B
done_bad:
        mov     ax, -1                  ; Indicate we had a problem
        jmp     short done

@@:     xchg    si, dx                  ; SI -> sprite_list[i]
        mov     [si].next_node, dx
        cmp     bx, offset first_sprite
        jne     @F
        mov     [bx], si
        jmp     short done_good
@@:     mov     [bx].next_node, si
done_good:
        xor     ax, ax
done:   ret

insert_sprite endp
; ***************************************************************************
; void ADD_SPRITE( sprite_structure far *DEST, far *SOURCE )
; Adds a self-relative sprite motion to the end of a sprite circle
; Given:
;       DEST -> sprite circle header
;       SOURCE -> sprite to add on
; Returns:
;       SOURCE is added to the end of the sprite linked list and the
;       circle ends are rejoined, may the circle be unbroken (ie Johnny Cash)

add_sprite proc uses si di ds, dest:dword, source:dword

        lds     si, dest
        les     di, source
next_sprite:
        mov     ax, [si].animate_ptr    ; See if this is the end of the line
        cmp     ax, word ptr dest
        jne     @F
        mov     ax, [si].animate_ptr+2
        cmp     ax, word ptr dest+2
        je      got_the_end
@@:     lds     si, [si].animate_ptr
        jmp     next_sprite

got_the_end:
        mov     [si].animate_ptr, di
        mov     [si].animate_ptr+2, es
        mov     ax, word ptr dest
        mov     es:[di].animate_ptr, ax
        mov     ax, word ptr dest+2
        mov     es:[di].animate_ptr+2, ax

        ret
add_sprite endp

        end
```

End Listing Two

(continued on page 92)

Listing Three *(Listing continued, text begins on page 16.)*

```
/*****************************************************************
        TITLE:  SPRITES.C
        Displays a sprite file on an EGA screen
        Written by: Rahner James, CS
                    of Ryu Consulting, Inc.
*****************************************************************/
#include <stdio.h>
#include <dos.h>
#include <fcntl.h>
/*****************************************************************
                VARIOUS DEFINITIONS
*****************************************************************/
#pragma pack(1)

typedef unsigned char   uc;
typedef unsigned int    ui;
typedef unsigned long   ul;
/*****************************************************************
                EXTERNAL DECLARATIONS
*****************************************************************/
extern  void ega_convert
();
extern  ui ega_calculate( uc far * );
extern  void ega_install();
extern  void ega_clear_area( ui, ui, ui, ui );
/*****************************************************************
                GLOBAL DATA
*****************************************************************/
ui      min_x=0, min_y=0, max_x=639, max_y=199;
/*****************************************************************
        long READ_ALL_FILE( uc *FILENAME, uc huge *BUFFER, ul BUFFER_SIZE )
        Opens and reads an entire sprite file
        Given:
                FILENAME -> name of sprite file to read
                BUFFER -> buffer to read the sprite file into
                BUFFER_SIZE = number of bytes the buffer can hold
        Returns:
                File is opened, read and closed
                Number of bytes read, if all went well
                If error, returns -1
*****************************************************************/
long read_all_file( uc *filename, uc huge *buffer, ul buffer_size )
{
        long    rv = 0;
        ui      handle, dos_return, amount_read;
        ui      amount_to_read;
        if ( _dos_open( filename, O_RDONLY, &handle ) )
                return -1;
        while ( buffer_size )
        {
                amount_to_read = buffer_size<60000L ? buffer_size : 60000L;
                if ( _dos_read( handle, buffer+rv, amount_to_read, &amount_read ) )
                {
                        rv = -1;
                        break;
                }
                rv += amount_read;
                if ( amount_read < 60000 )
                        break;
                buffer_size -= amount_read;
        }
        _dos_close( handle );
        return rv;
}

/*****************************************************************
        void DO_BACKGROUND( void )
        Sets up the background for the sprite visual screen
        Given:
                nothing
        Returns:
                visual sprite screen erased
*****************************************************************/
void do_background( void )
{
        ega_clear_area( min_x, min_y, max_x, max_y );
}
/*****************************************************************
        uc SET_MODE( uc MODE_NUMBER )
        Sets the video mode
        Given:
                Mode number to set video to
        Returns:
                Present video mode number
*****************************************************************/
uc set_mode( uc mode_number )
{
        uc rv;
        union REGS regs;

        regs.h.ah = 15;
        int86( 0x10, &regs, &regs );
        rv = regs.h.al;

        regs.h.ah = 0;
        regs.h.al = mode_number;
        int86( 0x10, &regs, &regs );
        return rv;
}
/*****************************************************************
        MAIN( int ARGC, uc *ARGV[] )
        Allocates memory, reads in a sprite file, displays the sprites
        until a key is pressed, frees up memory and interrupt vectors
        Given:
                ARGC = number of command line values, must be > 1
```

```
                ARGV[1] -> file name of the sprite file to display
        Returns:
                0 if all went well, otherwise numbered according to error
*************************************************************************** /
main( int argc, uc *argv[] )
{
        ui      i, x, y;
        uc      far *file_ptr, huge *sprite_start, huge *buffer_start;
        uc      huge *sprite_ptr[20];
        uc      old_mode;
        ui      memory_segment;
        ul      memory_size=0, file_size;
/* Check initial values and allocate memory for buffers */
        if ( argc<2 )
        {
                printf( "\nNo file name has been given\n" );
                exit( 1 );
        }

        if ( _dos_allocmem( -1, (ui *)&memory_size ) )
        {
                if ( _dos_allocmem( memory_size, &memory_segment ) )
                {
                        printf( "\nMemory allocation error\n" );
                        exit( 2 );
                }
        }
        else
        {
                memory_segment = memory_size;
                memory_size = 0xffff;
        }
        memory_size <<= 4;
        buffer_start = (uc huge *)((ul)memory_segment << 16L);
/* Read in the sprite file and then convert it to our intenal structure */
        file_ptr = buffer_start;
        if ( (file_size=read_all_file(argv[1],file_ptr,memory_size)) == -1 )
        {
                _dos_freemem( memory_segment );
                printf( "\nGot error reading %s.  Aborting.\n", argv[1] );
                exit( 3 );
        }
        clear_sprite_list();
        sprite_start = file_ptr + file_size;
        for ( i=0 ; i<20 && file_size ; ++i )
        {
                ega_convert( sprite_ptr[i]=sprite_start, file_ptr );
                x = ega_calculate( file_ptr );
                sprite_start += x;
                x = (ui)*file_ptr * (ui)*(file_ptr+2) + 4;
                file_ptr += x;
                if ( file_size > (ul)x )
                        file_size -= (ul)x;
                else
                        file_size = 0;
        }
/* Create linked list of sprite circles */
        insert_sprite( 100,100, 100,100, 0,0, 7, sprite_ptr[0] );
        for ( x=1 ; x<i ; ++x )
                add_sprite( sprite_ptr[0], sprite_ptr[x] );
/* Setup the EGA screen mode and interrupt vector */
        old_mode = set_mode( 0x10 );
        ega_install();
/* Process the sprite list until someone taps a key */
        while ( !kbhit() )
                do_sprite_list();
/* Restore screen and allocated memory to original state */
        ega_rip_out();
        set_mode( old_mode );
        _dos_freemem( memory_segment );
        exit( 0 );
```

End Listing Three

Listing Four

```
sprite.obj:     sprite.c
        cl /c sprite.c

ega_drv.obj:    ega_drv.asm
        masm ega_drv;

gen_asm.obj:    gen_asm.asm
        masm gen_asm;

sprite.exe:     sprite.obj ega_drv.obj gen_asm.obj
        link sprite+gen_asm+ega_drv;
```

End Listings

Listing One *(Text begins on page 28.)*

```
/*-------------------------------------------------------------------------*/
/* dma.c -- subroutines for dma data acquisition
 *
 * The calling routine must declare the variables in the "extern" list
 * below, and the reset_irq() function. Communication from the main
 * program to the subroutines is mostly through these global variables.
 * The calling routine must give values to dma_chan, dma_irq and buf_size,
 * then call alloc_dma_buf(), dma_setup(), and start_dma(). As each
 * dma buffer fills up, the interrupt service routine calls start_dma() on
 * the next buffer. The calling routine can wait for buf_index to
 * change, then process data pointed to by curr_buf. Cleanup is done
 * by dma_finish(), which is called automatically when the program exits.
 *
 * Compiler: Microsoft C Version 5.0
 *           Set /Gs switch to remove stack probes (a necessity for any
 *           function called at interrupt state!)
 *
 * Tom Nolan - 11/3/89
 */

#pragma check_stack(off)
#include <dos.h>

/* DMA Register Definitions */

#define DMA0_BASE   0x00            /* address of dma controller (chan 0-3) */
#define DMA1_BASE   0xC0            /* address of dma controller (chan 4-7) */

/* Interrupt Controller Definitions */

#define INTA00      0x20            /* base address of int ctrlr */
#define INTA01      0x21            /* address of int ctrlr 2nd reg */
#define EOI         0x20            /* code for non-specific end-of-int */


/* External Variables */

extern char far *dma_buffers[];     /* array containing buffer addresses */
extern int      buf_index;          /* index of current buffer in array */
extern char far *curr_buf;          /* pointer to just-filled buffer */
extern unsigned buf_size;           /* size of buffers in bytes */
extern int      lost_buffers;       /* count of buffers unable to be written
*/
extern int      dma_irq;            /* h/w interrupt when dma complete (0-7)
*/
extern int      dma_chan;           /* channel number for dma operation */
extern int      file_handle;        /* handle of archive file (0=no file) */
extern void     reset_irq();        /* function to reset interrupt request */


/* local variables - placed in static storage to
 * avoid excessive stack usage in interrupt routines */

static union  REGS  r;              /* general registers */
static struct SREGS s;              /* segment registers */
static int sel;                     /* dma channel select bits */
static int basereg;                 /* dma controller base address register */
static int cntreg;                  /* dma controller count register */
static int maskreg;                 /* dma controller mask register */
static int modereg;                 /* dma controller mode register */
static int pagereg;                 /* dma page address register */
static int page_tbl[] =             /* table of page register addresses */
        { 0x87, 0x83, 0x81, 0x82,   /* for dma channels 0, 1, 2, 3 */
          0x8f, 0x8b, 0x89, 0x8a }; /*                 4, 5, 6, 7 */
char far *dos_crit_addr;            /* address of DOS critical section flag */
static void                         /* space for saved int vector contents */
    (interrupt far *dma_int_save)();

/* macros for extracting bytes from 20-bit addresses */

#define LSB(x)  *((unsigned char *) &x)
#define MSB(x)  *(((unsigned char *) &x) + 1)
#define PAGE(x) *(((unsigned char *) &x) + 2)

/* Function Prototypes */

void dma_setup(void);
void dma_finish(void);
int alloc_dma_buf(void);
void start_dma(char far *, unsigned);
void interrupt far dma_isr(void);
int write_buf();

/*-------------------------------------------------------------------------*/
int alloc_dma_buf()                 /* allocate a pair of dma buffers */
{
    unsigned buf;                   /* temp variables for various */
    unsigned max;                   /*   paragraph addreses       */
    unsigned seg;
    unsigned size;                  /* buffer size in paragraphs */

    /* This routine allocates a pair of buffers that can be
     * filled by dma. The buffers are guaranteed to be
     * aligned so that they do not cross physical page boundaries.
     * Before calling this routine, set the value of buf_size to
     * the required number of bytes in each buffer. The maximum
     * buffer size is 64K bytes, which can be allocated
     * by specifying a buf_size of zero. The byte count is converted
     * to paragraphs, which are the units the DOS memory allocation
     * functions work with. Buffer addresses are returned in dma_buffers[0]
     * and dma_buffers[1]. The return value is zero if the allocation succeded,
     * non-zero (an MS-DOS error code) otherwise.
     */

    size = (buf_size == 0) ?        /* convert bytes to paragraphs */
              0x1000 : buf_size >> 4; /* ..by dividing by 16 */
    _dos_allocmem(0xffff, &max);    /* get max paragraphs from dos */
    _dos_allocmem(max, &seg);       /* now grab it all */
    buf = seg;                      /* initial attempt at buffer segment
*/
    if( ((buf + size - 1) & 0xf000)     /* if buffer crosses           */
            != (buf & 0xf000) )         /*     phys page bdry          */
        buf = (buf & 0xf000) + 0x1000;  /* ...adjust to next phys page */

    dma_buffers[0] = (char far *)       /* convert buffer segment          */
              ((long) buf << 16);       /* ... to far pointer for return   */

    buf += size;                        /* initial attempt at next buffer */
    if( ((buf + size - 1) & 0xf000)
            != (buf & 0xf000) )
        buf = (buf & 0xf000) + 0x1000;  /* adjust if it crosses page bdry */

    dma_buffers[1] = (char far *)       /* return it as a far pointer */
              ((long) buf << 16);

    size = buf + size - seg;            /* compute actual size needed */
    return                              /* free unneeded memory and     */
        _dos_setblock(size, seg, &max); /*   return error if not enough */
}

/*-------------------------------------------------------------------------*/
void dma_setup()                        /* set up for dma operations */
{
    /* Before calling this routine set the following variables:
     *
     *      dma_chan - channel number (hardware dependent)
     *      dma_irq  - interrupt request number 0-7 (hardware dependent)
     */

    int intmsk;

    sel  = dma_chan & 3;                /* isolate channel select bits */
    pagereg = page_tbl[dma_chan];       /* locate corresponding page reg
*/

    if(dma_chan < 4)                    /* setup depends on chan number */
    {
        basereg = DMA0_BASE + sel * 2;  /* standard dma controller    */
        cntreg  = basereg + 1;          /* note that this controller  */
        maskreg = DMA0_BASE + 10;       /*     is addressed on byte   */
        modereg = DMA0_BASE + 11;       /*          boundaries        */
    }
    else
    {
        basereg = DMA1_BASE + sel * 4;  /* alternate dma ctrlr (AT only)
*/
        cntreg  = basereg + 2;          /* note that this controller
*/
        maskreg = DMA1_BASE + 20;       /*     is addressed on word
*/
        modereg = DMA1_BASE + 22;       /*          boundaries
*/
    }

    r.h.ah = 0x34;                  /* dos "get critical flag addr" function
*/
    intdosx(&r, &r, &s);
    dos_crit_addr = (char far *)    /* save its address so it can be tested */
        (((long) s.es << 16) | r.x.bx); /* ... as a far pointer */

    if(dma_irq < 0 || dma_irq > 7)  /* validate interrupt number */
        return;
    dma_int_save =                  /* save current contents of dma int vec */
        _dos_getvect(dma_irq + 8);
    _dos_setvect(dma_irq+8, dma_isr); /* set up new int service routine */

    intmsk = inp(INTA01);           /* get current interrupt enable mask */
    intmsk &= ~(1 << dma_irq);      /* clear mask bit for dma interrupt */
    outp(INTA01, intmsk);           /* output new mask, enabling interrupt */
    atexit(dma_finish);             /* register exit function */
}

/*-------------------------------------------------------------------------*/
static void dma_finish()            /* called via atexit() mechanism */
{
    int intmsk;

    intmsk = inp(INTA01);           /* get current interrupt enable mask */
    intmsk |= (1 << dma_irq);       /* set mask bit for dma interrupt */
    outp(INTA01, intmsk);           /* output new mask, disabling interrupt */

    _dos_setvect(dma_irq+8, dma_int_save);  /* restore old vector contents */
}

/*-------------------------------------------------------------------------*/
void start_dma(buf, count)          /* start a dma operation */
char far *buf;                      /* address of buffer to be filled */
unsigned count;                     /* size of buffer in bytes */
{
    int page;
    unsigned long addr =            /* 20-bit address of dma buffer */
            FP_OFF(buf) +
            ((long) FP_SEG(buf) << 4);

    /* This routine starts a dma operation. It needs to know:
     *      - the address where the dma buffer starts;
     *      - the number of bytes to transfer;
     * The dma buffer address is supplied in segmented, far-pointer
     * form (as returned by alloc_dma_buf()). In this routine it is
     * converted to a 20-bit address by combining the segment and
     * offset. The upper four bits are known as the page number, and
     * are handled separately from the lower 16 bits. The transfer
     * count is decremented by 1 because the dma controller reaches
     * terminal count when the count rolls over from 0000 to ffff.
     *
     * The dma transfer stops when the channel reaches terminal count.
```

(continued on page 96)

Listing One *(Listing continued, text begins on page 28.)*

```
     * The terminal count signal is turned around in the interface
     * hardware to produce an interrupt when dma is complete.
     *
     * Channels 4-7 are on a separate dma controller, available on
     * the PC-AT only. They perform 16-bit transfers instead of 8-bit
     * transfers, and they are addressed in words instead of bytes.
     * This routine handles the addressing requirements based
     * on the channel number.
     *
     * dma_setup() needs to be called before start_dma() in order to
     * assign values to maskreg, modereg, etc.
     */

    page = PAGE(addr);              /* extract upper bits of address */

    if(dma_chan >= 4)               /* for word-oriented channels... */
    {
        count >>= 1;                /* convert count to words */
        addr  >>= 1;                /* convert address to words */
        page  &=  0x7e;             /* address bit 16 is now in 'addr' */
    }

    count--;                        /* compute count-1 (xfr stops at ffff) */
    outp(maskreg,     sel | 0x04);  /* set mask bit to disable dma */
    outp(modereg,     sel | 0x44);  /* xfr mode (sngl, inc, noinit, write) */
    outp(basereg,     LSB(addr) );  /* output base address lsb */
    outp(basereg,     MSB(addr) );  /* output base address msb */
    outp(pagereg,     page      );  /* output page number to page register */
    outp(cntreg,      LSB(count));  /* output count lsb */
    outp(cntreg,      MSB(count));  /* output count msb */
    outp(maskreg,     sel       );  /* clear mask bit, enabling dma */
}

/*-------------------------------------------------------------------------*/
static void interrupt far dma_isr()
{
    /* This routine is entered upon completion of a dma operation.
     * At this point the current dma buffer is full and we can
     * write it to disk. We set the "available data" pointer
     * to point to the just-filled buffer, and start the next dma
     * operation on the other buffer. At the conclusion of
     * operations, we output a non-specific end-of-interrupt
     * to the interrupt controller.
     *
     * The PC bus provides no mechanism for "unlatching" an
     * interrupt request once it has been serviced. In order to
     * enable the next interrupt, the hardware must be designed
     * so that the request can be reset, by a write to an i/o
     * port, for example. The external routine reset_irq()
     * must be coded to perform this function.
     *
     * Declaring this routine as type 'interrupt', ensures
     * that all registers are saved, the C data segment is set
     * correctly, and that the routine returns with an IRET
     * instruction. Further interrupts are disabled during the
     * execution of this routine.
     */

    curr_buf = dma_buffers[buf_index];      /* post just-filled buf address */
    buf_index ^= 1;                         /* index next buffer */
    start_dma(dma_buffers[buf_index],       /* start dma on next buffer */
              buf_size);
    if( file_handle )                       /* if disk is enabled.. */
        write_buf();                        /* write buffer to disk */
    reset_irq();                            /* do hardware-specific reset */
    outp(INTA00, EOI);                      /* signal end of int */
}

/*-------------------------------------------------------------------------*/
static int write_buf()              /* write buffer to disk file */
{
    if( *dos_crit_addr )            /* first check dos critical section flag
*/
    {
        lost_buffers++;             /* ..if set, skip writing this buffer */
        return 0;                   /* ..not really an error in this case */
    }

    r.x.dx = FP_OFF(curr_buf);      /* ok to write now, set address in */
    s.ds   = FP_SEG(curr_buf);      /*   proper registers for dos call */
    r.x.bx = file_handle;           /* set file handle to write to */
    r.x.cx = buf_size;              /* set byte count for write */
                                    /* WARNING - can't write 64K! */
    r.h.ah = 0x40;                  /* dos write-to-file-handle function */
    if(intdosx(&r, &r, &s) == buf_size  /* check return value and.. */
            && r.x.cflag == 0)      /* ..carry flag for success code */
        return 0;                   /* return success */
    else
    {
        lost_buffers++;             /* didn't write this buffer */
        return 1;                   /* return failure */
    }
}
```

End Listing One

Listing Two

```
/*-------------------------------------------------------------------------
*/
/* test.c -- test dma data acquisition
 *
 * Compiler: Microsoft C Version 5.0
 * Compile with /Gs option (or use #pragma) because
 * reset_irq() is called from interrupt.
 *
 * Tom Nolan - 11/3/89
 */

#include <bios.h>
int far *dma_buffers[2];        /* pointers to two buffers */
int far *curr_buf;              /* pointer to current buffer */
int buf_size;                   /* buffer size */
int buf_index;                  /* index of current buffer */
int dma_irq = 3;                /* hardware int request line */
int dma_chan = 1;               /* hardware dma channel number */
int file_handle = 0;            /* file handle */
int lost_buffers = 0;           /* write errors */

/* In this program, each dma buffer will be filled with
 * NUMFR "frames", each of size FRSIZE (in words). The second
 * word of each frame is a frame counter, which increments
 * modulo 256. The program checks the frame counters to make
 * sure they are sequential and no data was lost.
 */

#define FRSIZE  64              /* words per frame */
#define NUMFR   8               /* frames per dma buffer */

/*-------------------------------------------------------------------------
*/
main()
{
    int temp;
    int i;
    unsigned char frame;
    int far *cp;

    reset_irq();                            /* clear interrupt request */
    buf_size = FRSIZE * NUMFR * sizeof(int); /* figure out buffer size */
    alloc_dma_buf();                        /* allocate buffers */
    printf("buf1 = %p buf2 = %p\n",         /* informational output */
        dma_buffers[0], dma_buffers[1]);
    dma_setup();                            /* set up for dma operations */

    outp(0x3a0,0);                          /* reset fifo on hw interface */
    start_dma(dma_buffers[0], buf_size);    /* start up the data acq */
    file_handle = creat("tmp.dat");         /* open a file for raw data */
    temp = buf_index;

    while( !_bios_keybrd(_KEYBRD_READY) )   /* quit on next keystroke */
    {
        if(temp != buf_index)               /* wait for dma complete */
        {
            printf("%d: ",temp);            /* print buffer index */

            for(i=0, cp = curr_buf+1; i<NUMFR; i++, cp += FRSIZE)
            {
                if(frame != *cp) printf("*"); /* frame counter was bad */
                else printf(" ");           /* frame counter was good */
                printf("%04x", *cp);        /* print frame counter */
                frame = *cp + 1;            /* next expected counter value
*/
            }
            printf(" : %d\n",lost_buffers); /* keep track of lost writes */
            temp = buf_index;               /* next expected buffer number
*/
        }
    }
    close(file_handle);                     /* close data file */
    exit(0);                                /* halt dma and exit */
}

#pragma check_stack(off)

/*-------------------------------------------------------------------------
*/
reset_irq()             /* clear interrupt request in hardware */
{
    inp(0x3A0);
}
```

End Listings

Listing One *(Text begins on page 38.)*

```
SCREEN 0
\ ZEN, Version 1.60--a simple classical Forth ZEN 1.60 is a model implementa-
tion of the unofficial ANS Forth with Double-Number, File Access, and BLOCK
Standard Extensions (BASIS6). This model is not endorsed by the ANS X3J14 com-
mittee. { Comments to go back to the ANS committee look like this.} ZEN 1.60
generates an IBM PC 64K small-model ROM-able nucleus. BX register is top-of-
stack. DTC with JMP code field. Assumes segment registers CS = DS = ES Thanks
to Wil Baden for his suggestions. This is a working document. No guarantees
are made to its accuracy or fitness. While this is a working document, it is
copyrighted 1989 by Martin J. Tracy. All rights are reserved.

SCREEN 1
\ ZEN nucleus
FORTH DEFINITIONS  8 K-OF-ROM !  " KERNEL.COM" MAKE-OBJECT

32  CONSTANT #Jot     ( number conversion area in bytes)
128 CONSTANT #Safe    ( CREATE safety area--   in bytes)
128 CONSTANT #User    ( total user area size-- in bytes)

HEX
0100 BFFF 2DUP 2CONSTANT #ROM  ROMORG 2!  ( start & end of ROM)
C000 FFFF 2DUP 2CONSTANT #RAM  RAMORG 2!  ( start & end of RAM)
0000 #User - CONSTANT #RP0  ( top of return stack)
#RP0  0080 - CONSTANT #SP0  ( top of data   stack)

START   DECIMAL 2 LOAD   FINIS

SCREEN 2
\ Main LOAD screen
HERE EQU Power  2 CELLS ( power-up) GAP   ," C 1989 by M Tracy"
HERE EQU D0   #ROM , ,  #RAM , ,
HERE EQU H0 ( h) 0 ,    HERE EQU F0  0 , 0 ,  ( forth vlink)
HERE EQU T0 ( r) 0 ,    HERE EQU S0   #SP0 ,

          7 17 THRU  ( Kernel primitives)
         19 27 THRU  ( Numbers and I/O)
         29 41 THRU  ( Interpreter)
         43 69 THRU  ( Compiler)
         71 75 THRU  ( Device dependencies)
         81 86 THRU  ( Mass storage extension)
         77 79 THRU  ( Initialization)
                     ( Application, if any)

HERE H0 !   THERE T0 !

SCREEN 3
\ Documentation requirements
ZEN 1.60 supports Double-Number, File Access, and BLOCK Standard Extensions.
To compile the BLOCK extension, load the two screens following the File Access
extension. There are two 8-bit bytes per cell. Counted strings may be as long
as 255 bytes. Division is rounded-down. To change to floored division, load
two screens following the mixed-precision rounded-down operators. The system
dictionary is approximately 7K address units (au's) leaving 56K for the ap-
plication. {#RAM & #ROM are currently set for 40K of application dictionary
and 16K of RAM.} {The data stack grows downwards towards the bottom of RAM.}
{The return stack is currently set for 128 au's of RAM.} Only dumb (glass)
terminals are supported. { How are minimum facilities to be specified?}

SCREEN 4
\ Errors and exceptions
If the input stream is inadvertantly exhausted: ABORT" ?"
If a word is not found: ABORT" ?"
If control structures are incorrectly nested: ABORT" Unbalanced"
If insufficient space in the dictionary: ABORT" No Room"
If insufficient number of stack entries: ABORT" Stack?"
If FORGETing within the nucleus: ABORT" Can't"

Division by zero returns a quotient of zero and a remainder equal to the divi-
dend. Data and return stack overflows are not detected: the system may crash
or hang, if you are lucky. Execution of compiler words while interpreting is
not prevented; the result of such execution is undefined. Invalid and out-of-
range arguments are not checked: the result of using such arguments is very
undefined.

SCREEN 5
\ Key to auxiliary commands
Several words used by the metacompiler are described here.

¦               make the next word headerless.
," ccc"         compile the characters "ccc."

a ORG           reset HERE to address a.
n EQU <name>    equivalent to a headerless constant with value n.
LABEL <name>    equivalent to  HERE EQU <name> but also activates
                the CODE assembler.
CODE <name>     begins a machine-code definition, usually ended
                by  END-CODE  or  C;

I> and >I       like R> and >R when used to get return addresses.

BASE is returned to DECIMAL after each block is LOADed.

SCREEN 6
 ------------------------------------------------------------
¦                                                            ¦
¦  Please direct all comments and inquiries to Martin Tracy  ¦
¦                                                            ¦
¦                                                            ¦
 ------------------------------------------------------------

SCREEN 7
\ ------ Kernel primitives -------------------------
LABEL colon   BP DEC  BP DEC   SI 0 [BP] MOV   SI POP   NEXT
\ save I register on return stack and set it to new position.
\ This is the action of the code field in all colon definitions.

CODE EXIT   NOP
¦ CODE semi   0 [BP] SI MOV   BP INC  BP INC
¦ CODE nope   NEXT C;
\ semi is the action of the semicolon in all colon definitions.
\ EXIT differs from semi as an aid to decompilation.
\ nope is a "no operation" word used for initialization.

SCREEN 8
\ Data objects
LABEL addr  \ the action of all CREATEs.
   BX PUSH   3 # AX ADD  BX AX XCHG   NEXT

LABEL con   \ the action of all CONSTANTs and VARIABLEs.
   BX PUSH   3 # AX ADD  BX AX XCHG   0 [BX] BX MOV   NEXT C;

VARIABLE u  { Private}  \ USER area pointer.
LABEL uvar  \ the action of all USER variables.
   BX PUSH   3 # AX ADD   BX AX XCHG
   0 [BX] BX MOV   u ) BX ADD   NEXT

LABEL (does)   BP DEC  BP DEC  SI 0 [BP] MOV  \ run-time DOES>
   SI POP  BX PUSH   3 # AX ADD   BX AX XCHG    NEXT C;

SCREEN 9
\ Stack manipulation
CODE DUP  ( w - w w)   BX PUSH   NEXT C;
CODE DROP ( w)         BX POP    NEXT C;

CODE SWAP ( w w2 - w2 w)
   SP DI MOV              BX 0 [DI] XCHG   NEXT C;
CODE OVER ( w w2 - w w2 w)
   SP DI MOV   BX PUSH   0 [DI] BX MOV    NEXT C;

CODE ROT ( w w2 w3 - w2 w3 w)
   DX POP  AX POP   DX PUSH  BX PUSH   AX BX MOV   NEXT C;

CODE PICK ( w[u]...w[1] w[0] u - w[u]...w[1] w[0] w[u])
\ copy kth item to top of stack.
   BX SHL   SP BX ADD   0 [BX] BX MOV   NEXT C;

SCREEN 10
\ Memory access
CODE @ ( a - w)   0 [BX] BX MOV        NEXT C;
CODE ! ( w a)     0 [BX] POP  BX POP   NEXT C;

CODE C@ ( a - b)  0 [BX] BL MOV   BH BH SUB        NEXT C;
CODE C! ( b a)    AX POP  AL 0 [BX] MOV   BX POP   NEXT C;

CODE CMOVE ( a a2 u)
\ move count bytes from from to to, leftmost byte first.
   BX CX MOV   SI BX MOV   DI POP  SI POP
   REP BYTE MOVS   BX SI MOV   BX POP   NEXT C;

SCREEN 11
\ Math operators
¦ CODE tic   NOP
¦ CODE lit   WORD LODS   BX PUSH  AX BX MOV    NEXT C;
\ push the following (in-line) number onto the stack.

CODE + ( n n2 - n3)   AX POP  AX BX ADD            NEXT C;
CODE - ( n n2 - n3)   AX POP  AX BX SUB   BX NEG   NEXT C;

CODE NEGATE ( n -  n2)   BX NEG   NEXT C;
CODE ABS    ( n - +n2)
   BX BX OR  1 L# JNS   BX NEG   1 L: NEXT C;

CODE +! ( n a)   AX POP   AX 0 [BX] ADD  BX POP   NEXT C;
\ increment number at address by n.

SCREEN 12
\ Math and logical
CODE 1+ ( n - n2)   BX INC   NEXT C;
CODE 1- ( n - n2)   BX DEC   NEXT C;

CODE 2* ( n - n2)   BX SHL   NEXT C;  { CONTROLLED} { Require?}
CODE 2/ ( n - n2)   BX SAR   NEXT C;  ( arithmetic)

CODE AND ( m m2 - m3)   AX POP  AX BX AND    NEXT C;
CODE OR  ( m m2 - m3)   AX POP  AX BX OR     NEXT C;
CODE XOR ( m m2 - m3)   AX POP  AX BX XOR    NEXT C;

CODE NOT ( w - w2)   BX NOT   NEXT C;  { ( m - m2) ?}

SCREEN 13
\ Comparisons
CODE 0 ( - n)      BX PUSH   BX BX SUB    NEXT C;  { Feature}
CODE 1 ( - n)      BX PUSH   1 # BX MOV   NEXT C;  { Feature}
CODE TRUE ( - m)   BX PUSH  -1 # BX MOV   NEXT C;  { Control?}

CODE = ( n n2 - f)   AX POP  AX BX CMP
   TRUE # BX MOV  1 L# JZ   BX INC   1 L: NEXT C;
CODE <  ( n n2 - f)   AX POP  BX AX SUB
   TRUE # BX MOV  1 L# JL   BX INC   1 L: NEXT C;
CODE U< ( u u2 - f)   AX POP  BX AX SUB
   TRUE # BX MOV  1 L# JB   BX INC   1 L: NEXT C;

: > ( n n2 - f)   SWAP < ;

SCREEN 14
\ Comparisons against zero and CELL operators
CODE 0= ( n - f)
   BX BX OR  TRUE # BX MOV  1 L# JZ   BX INC   1 L: NEXT C;
CODE 0< ( n - f)
   BX BX OR  TRUE # BX MOV  1 L# JS   BX INC   1 L: NEXT C;

: 0> ( n - f)   0 > ;


2 CONSTANT CELL  { Feature}

CODE CELL+ ( a - a2)   BX INC  BX INC   NEXT C;
CODE CELLS ( a - a2)   BX SHL           NEXT C;
```

(continued on page 100)

Listing One *(Listing continued, text begins on page 38.)*

```
SCREEN 15
\ Branches and loops
: CODE  branch          \ unconditional branch.
   0 [SI] SI MOV   NEXT C;
: CODE ?branch ( f)     \ branch if zero.
   BX BX OR  BX POP     ' branch JZ   2 # SI ADD   NEXT C;

: CODE (do) ( n n2)     \ begin DO...LOOP structure.
   4 # BP SUB       AX POP    HEX 8000 DECIMAL # AX ADD
   AX 2 [BP] MOV    AX BX SUB    BX 0 [BP] MOV    BX POP    NEXT C;

: CODE (loop)           \ terminate DO...LOOP structure.
   WORD 0 [BP] INC   ' branch JNO
LABEL  >loop    2 # SI ADD
: CODE >undo    4 # BP ADD    NEXT
: CODE (+loop) ( n)     \ terminate DO...+LOOP structure.
   BX 0 [BP] ADD  BX POP   ' branch JNO  >loop JO    NEXT C;

SCREEN 16
\ Return stack
CODE >R ( w)     BP DEC  BP DEC  BX 0 [BP] MOV    BX POP   NEXT C;

CODE R@ ( - w)   BX PUSH  0 [BP] BX MOV    NEXT C;
CODE I  ( - n)   BX PUSH  0 [BP] BX MOV  2 [BP] BX ADD    NEXT C;
CODE J  ( - n)   BX PUSH  4 [BP] BX MOV  6 [BP] BX ADD    NEXT C;
CODE R> ( - w)   BX PUSH  0 [BP] BX MOV  BP INC  BP INC   NEXT C;

CODE 2>R ( w w2)
\ push w and w2 to the return stack, w2 on top.
   4 # BP SUB    BX 0 [BP] MOV    2 [BP] POP    BX POP      NEXT C;

CODE 2R> ( - w w2)
\ pop w and w2 from the return stack.
   BX PUSH   2 [BP] PUSH    0 [BP] BX MOV    4 # BP ADD    NEXT C;

SCREEN 17
\ Optimizations and EXECUTE
CODE NIP  ( w w2 - w2)        { CONTROLLED}  AX POP    NEXT C;
CODE TUCK ( w w2 - w2 w w2)  { CONTROLLED}  AX POP
   BX PUSH  AX PUSH    NEXT C;

CODE ?DUP ( w - w w : 0 - 0)
   BX BX OR  1 L# JZ    BX PUSH  1 L: NEXT C;

CODE EXECUTE ( w)    BX AX XCHG    BX POP    AX JMP C;

CODE @EXECUTE ( w)  { Control?}  { Why w and not a?}
\ @EXECUTE is equivalent to @ EXECUTE but is much faster.
   BX DI MOV    BX POP    0 [DI] AX MOV    AX JMP C;

SCREEN 18



SCREEN 19
\ ------ Input/Output -----------------------------
\ In ZEN, consecutive headerless variables form a category
\ which can be extended but not reduced or reordered.

0 USER entry  2 CELLS + ( skip multitasking hooks)
  USER r       : USER SP0
  USER x       \ XFER vector pointer.
  USER BASE    : USER dpl     : USER hld    EQU #I/O

: THERE ( - a)    r @ ;  { ROM}
: PAD   ( - a)    r @ [ #Jot ] LITERAL + ;  { CONTROLLED}
{ pictured number staging area size undefined?}

: DECIMAL    10 BASE ! ;
: HEX        16 BASE ! ;  { CONTROLLED}

SCREEN 20
\ Double-value data stack operators
CODE 2DUP ( w w2 - w w2 w w2)    SP DI MOV    BX PUSH
   0 [DI] PUSH    NEXT C;

CODE 2DROP ( w w2)    BX POP  BX POP    NEXT C;

CODE 2SWAP ( w w2 w3 w4 - w3 w4 w w2)    AX POP  CX POP  DX POP
   AX PUSH  BX PUSH  DX PUSH  CX BX MOV    NEXT C;
: 2OVER ( d d2 - d d2 d)        2>R 2DUP  2R> 2SWAP ;
: 2ROT  ( d d2 d3 - d2 d3 d)    2R> 2SWAP 2R> 2SWAP ;
{ CONTROLLED}  { Require?}

CODE 2@ ( a - w w2)    2 [BX] PUSH  0 [BX] BX MOV        NEXT C;
CODE 2! ( w w2 a)      0 [BX] POP   2 [BX] POP    BX POP  NEXT C;

SCREEN 21
\ Numeric conversion math support
CODE D+ ( d d2 - d3)    AX POP    DX POP    CX POP
   AX CX ADD    CX PUSH     DX BX ADC    NEXT C;

CODE DNEGATE ( d - d2)    AX POP  AX NEG  AX PUSH
   0 # BX ADC    BX NEG    NEXT C;

: MAX ( n n2 - n3)    2DUP <     IF  SWAP  THEN  DROP ;
: MIN ( n n2 - n3)    2DUP < 0= IF  SWAP  THEN  DROP ;


SCREEN 22
\ Numeric conversion math support
CODE UM* ( u u2 - ud)
   AX POP    BX MUL     AX PUSH    DX BX MOV    NEXT C;

CODE UM/MOD ( ud u - u2 u3)
\ return rem u2 and quot u3 of unsigned ud divided by u.
\ On zero-divide, return quot=0 and rem=low-word-of-ud.
   DX POP  AX AX SUB  BX DX CMP  1 L# JAE
   AX POP  BX DIV  DX PUSH        1 L: AX BX MOV    NEXT C;

SCREEN 23
\ Input number conversion
ASCII A  ASCII 9  1+ - EQU A-10

: : digit ( c base - n t : ? 0)
\ true if the char c is a valid digit in the given base.
   SWAP [ASCII] 0 -  9 OVER <  DUP
   IF  DROP  A-10 -  10  THEN
   >R  DUP R@ -  ROT R> -  U< ;

: CONVERT ( +d a - +d2 a2)
\ convert the char sequence at a+1 and accumulate it in +d.
\ a2 is the address of the first non-convertable digit.
   BEGIN  1+ DUP >R  C@ BASE @ digit
   WHILE  SWAP  BASE @ UM* DROP  ROT  BASE @ UM*  D+  R>
   REPEAT DROP  R> ;

SCREEN 24
\ Output number conversion
: <#   PAD hld ! ;
: #> ( wd - a u)    2DROP hld @ PAD OVER - ;

: HOLD ( c)    TRUE hld +!  hld @ C! ;
\ add character c to output string.
: SIGN ( n)    0< IF  [ASCII] - HOLD  THEN ;
\ add "-" to output string if w is negative.

: # ( ud - ud2)
\ transfer the next digit of ud to the output string.
   BASE @ >R  0 R@ UM/MOD  R> SWAP >R  UM/MOD  R>
   ROT 9 OVER < IF  A-10 + THEN  [ASCII] 0 + HOLD ;

: #S ( ud - ud2)    BEGIN  #  2DUP OR  0= UNTIL ;
\ convert all remaining digits of ud.  ud2 is 0 0 .

SCREEN 25
\ Transfers
LABEL xvar  \ the action of all transfers.
   u ) DI MOV    x [DI] DI MOV    3 # AX ADD    DI AX XCHG
   0 [DI] DI MOV    AX DI ADD    0 [DI] AX MOV    AX JMP C;

0 XFER TYPE ( a u)              XFER CR
  XFER KEYS ( a u) { Private}   XFER KEY? ( - f) { Extend?}
  XFER MARK ( a u) { Extend?}   XFER PAGE        { Extend?}
  XFER TAB ( n n2) { Extend?} ( Reserved)  DROP

\ KEYS is a simple unfiltered EXPECT which doesn't echo.
\ KEY? is true if a key is available.
\ MARK is like TYPE but highlights if possible.
\ PAGE clears the screen.
\ TAB  moves the cursor to the x (n) and y (n2) coordinates.

SCREEN 26
\ Print spaces
32 CONSTANT BL   { CONTROLLED}    \ ASCII blank

     HERE ( *) BL ,
: SPACE    ( *) LITERAL 1 TYPE ;

HERE ( * )  BL C, BL C, BL C, BL C, BL C, BL C, BL C, BL C,
: SPACES ( +n )   \ output w spaces.   Optimized for TYPE.
   ( * ) LITERAL  OVER 2/ 2/ 2/   ?DUP
   IF  0 DO  DUP 8 TYPE  LOOP  THEN  SWAP 7 AND TYPE ;

SCREEN 27
\ Print numbers
: (d.) ( d - a u)    \ convert a double number to a string.
   TUCK  DUP 0< IF  DNEGATE  THEN  <#  #S ROT SIGN  #> ;

: D. ( d)    (d.) TYPE SPACE ;
: U. ( u)         0 D. ;
:  . ( n)    DUP 0< D. ;

SCREEN 28

SCREEN 29
\ ------ Interpreter -------------------------------
#I/O ( continued from I/O layer)
  USER BLK  { BLOCK) { Require?}   USER >IN    \ keep together.
  USER #TIB   CELL+   \ #TIB and TIB's value.
  USER SPAN
  USER STATE  EQU #Used

  VARIABLE last    CELL ALLOT      \ last lfa and cfa.
: VARIABLE scr     CELL ALLOT      \ last error location.
: VARIABLE bal   : VARIABLE leaf   \ see compiler.

VARIABLE CONTEXT   { CONTROLLED}
VARIABLE CURRENT   { CONTROLLED}

: TIB ( - a)    #TIB CELL+ @ ;

SCREEN 30
\ Automatic variables
\ These variables are automatically initialized; see COLD.
VARIABLE h    : VARIABLE f    CELL ( ie vlink) ALLOT

  VARIABLE 'pause     \ multitasking hook.
: VARIABLE 'expect    \ deferred EXPECT
: VARIABLE 'source    \ deferred input stream.
: VARIABLE 'warn      \ redefinition warning.
: VARIABLE 'loc       \ source location field.
: VARIABLE 'val?      \ string to number conversion.

: VARIABLE key'   CELL ALLOT    \ one-key look-ahead buffer.

: HERE ( - a)    h @ ;

SCREEN 31
( String operators-- high-level definitions )  EXIT
: COUNT ( a - a2 u)    DUP C@ SWAP 1+ ;
```

```
\ transform counted string into text string.
: /STRING ( a u n - a2 u2)  { Control?}  ROT OVER + ROT ROT - ;
\ truncate leftmost n chars of string.  n may be negative.

: SKIP ( a u b - a2 u2)  { Control?}
\ return shorter string from first position unequal to byte.
   >R  BEGIN  DUP
       WHILE  OVER C@ R@ - IF  R> DROP  EXIT  THEN  1 /STRING
       REPEAT   R> DROP ;
: SCAN ( a u b - a2 u2)  { Control?}
\ return shorter string from first position equal to byte.
   >R  BEGIN  DUP
       WHILE  OVER C@ R@ =  IF  R> DROP  EXIT  THEN  1 /STRING
       REPEAT   R> DROP ;
```

SCREEN 32

```
( String operators-- low-level definitions )
CODE COUNT ( a - a2 u)   BX AX MOV   AX INC
\ transform counted string into text string.
   0 [BX] BL MOV   BH BH SUB   AX PUSH   NEXT C;
CODE /STRING ( a u n - a2 u2)  { Control?}  CX POP  AX POP
\ truncate leftmost n chars of string.  n may be negative.
   BX AX ADD  BX CX SUB   CX BX MOV   AX PUSH   NEXT C;

CODE SKIP ( a u b - a2 u2) { Contr?}  BX AX MOV  CX POP  DI POP
\ return shorter string from first position unequal to byte.
   1 L# JCXZ   REPE BYTE SCAS   1 L# JZ    CX INC  DI DEC
   1 L: DI PUSH  CX BX MOV   NEXT C;
CODE SCAN ( a l b - a2 u2) { Contr?}  BX AX MOV  CX POP  DI POP
\ return shorter string from first position equal to byte.
   1 L# JCXZ   REPNE BYTE SCAS   1 L# JNZ   CX INC  DI DEC
   1 L: DI PUSH  CX BX MOV   NEXT C;
```

SCREEN 33

```
\ More string operators
CODE FILL ( a u b)   \ store u b's, starting at addr a.
   BX AX MOV  CX POP  DI POP   REP BYTE STOS   BX POP   NEXT C;

: -TRAILING ( a +n - a2 +n2)   2DUP
\ alter string to suppress trailing blanks.
   BEGIN  2DUP         BL SKIP  DUP
   WHILE  2SWAP 2DROP  BL SCAN  REPEAT  2DROP  NIP - ;

EXIT
: FILL ( a u b)   \ store u b's, starting at addr a.
   SWAP ?DUP 0= IF  2DROP EXIT  THEN
   >R OVER C!  DUP 1+ R> 1- CMOVE ;
```

SCREEN 34

```
\ Input stream operators
¦ : source ( - a u)   #TIB 2@ ;   \ input stream source.
:  /source ( - a u)   'source @EXECUTE  >IN @ /STRING ;

¦ : accept ( n f)   IF  1+  THEN  >IN +! ;
\ accept characters by incrementing >IN.

: parse ( c - a u)   \ parse a character-delimited string.
   >R  /source  OVER SWAP  R> SCAN  >R  OVER -  DUP R> accept ;

: WORD ( c - a)      \ parse a character-delimited string;
\ leading delimiters are accepted and skipped;
\ the string is counted and followed by a blank (not counted).
   >R  /source  OVER R> 2>R  R@ SKIP  OVER SWAP  R> SCAN
   OVER R> -  SWAP accept  OVER -  31 MIN  THERE  DUP >R
   2DUP C!  1+ SWAP CMOVE  BL R@ COUNT + C!  R> ;
```

SCREEN 35

```
\ Dictionary search
CODE thread ( a w - a 0 , cfa -1 , cfa 1)
\ search vocabulary for a match with the packed name at  a .
   DX POP  SI PUSH
 1 L: 0 [BX] BX MOV                ( chain thru dictionary )
   BX BX OR  5 L# JZ               ( jump if end of thread )
   DX DI MOV  ( 'string)   BX SI MOV  2 # SI ADD   ( SI=nfa)
   0 [SI] CL MOV     31 # CX AND   0 [DI] CL CMP   ( count = ?)
   1 L# JNZ ( lengths <>) DI INC  SI INC  ( to body of 'string)
   REPE BYTE CMPS ( names =?)  1 L# JNZ   ( jump not matched)
   CX POP   SI PUSH  ( cfa )
   CX SI MOV   BYTE 32 # 2 [BX] TEST  ( immediate bit )
   TRUE # BX MOV  4 L# JZ   BX NEG     4 L: NEXT
 5 L: SI POP   DX PUSH ( 'str)  ( BX = 0)   NEXT C;
```

SCREEN 36

```
\ FIND  [ and ]
: FIND ( a - a 0 ¦ a - w -1 ¦ a - w 1)
\ search dictionary for a match with the packed name at  a .
\ Return execution address and -1 or 1 ( IMMEDIATE) if found;
\ ['] EXIT 1 if  a  has zero length;  a 0  if not found.
   DUP C@ ( a l) DUP
   IF  31 MIN OVER C! ( a) CONTEXT @ thread ( a -1/0/1) DUP
      IF  EXIT  THEN       CONTEXT @ f -
      IF  DROP  f thread  THEN   EXIT
   THEN ( a 0) 2DROP  ['] EXIT 1 ;

: ]  TRUE STATE ! ;  \ stop interpreting; start compiling.
: [     0 STATE ! ;  \ stop compiling; start interpreting.
  IMMEDIATE
```

SCREEN 37

```
\ Data and return stack
\ Set data and return stack pointers, respectively:
¦ CODE sp! ( a)   BX SP MOV  BX POP   NEXT C;
¦ CODE rp! ( a)   BX BP MOV  BX POP   NEXT C;

: RESET  { Feature}  \ reset return stack for error recovery.
   I>  entry CELL - rp!  >I ;
: PRESET { Feature}  \ empty both stacks and prepare system.
   SP0 @ sp!  I> entry rp! >I  SP0 @ 0 #TIB 2!  0 STATE ! ;

¦ : err   RESET ;
```

(continued on page 102)

Listing One *(Listing continued, text begins on page 38.)*

```
CODE DEPTH ( - n)   \ # items on stack before DEPTH is executed.
   BX PUSH   u ) BX MOV   SP0 [BX] BX MOV   SP BX SUB   BX SAR
   NEXT C;

SCREEN 38
( Memory management-- high-level definitions)  EXIT
: ALLOT ( n)   r +! ;   \ allocate n RAM data bytes.
: GAP   ( n)   h +! ;   \ allocate n dictionary bytes.  ( ROM)

: C, ( w)   h @ C!  1 h +! ;     \ ie  HERE C!  1 GAP ;
\ append low byte of w onto the dictionary.
: ,  ( w)   h @ !  CELL h +! ;   \ ie  HERE !  CELL GAP ;
\ append w onto the dictionary.

EXIT  ( In an all-RAM system:)
: GAP   ALLOT ;   : THERE   HERE ;    : >DATA   >BODY ;
: GOES>  [COMPILE] DOES> ;  IMMEDIATE

SCREEN 39
( Memory management-- low-level definitions)
CODE ALLOT ( n)   \ allocate n RAM data bytes.
   r # DI MOV   u ) DI ADD   BX 0 [DI] ADD  BX POP   NEXT C;
CODE GAP   ( n)   \ allocate n dictionary bytes.  ( ROM)
   h # DI MOV                BX 0 [DI] ADD  BX POP   NEXT C;

CODE C, ( w)   h # DI MOV   0 [DI] DI MOV
\ append low byte of w onto the dictionary.
   BL 0 [DI] MOV   1 # BX MOV  ' GAP JU
CODE ,  ( w)   h # DI MOV   0 [DI] DI MOV
\ append w onto the dictionary.
   BX 0 [DI] MOV   2 # BX MOV  ' GAP JU  FORTH

SCREEN 40
\ Code and data fields
: >BODY ( w - a)   3 + ;
: >DATA ( w - a)   3 + @ ;  ( ROM)

: >code ( cfa - 'code)  1+  DUP @ CELL+ + ;
\ finds code address associated with cfa.
: : alter ( 'code cfa)   1+  TUCK CELL+ -  SWAP ! ;
\ point the cf to the given code addr.  Skip the CALL byte.

: : nest, ( 'code )   HERE  232 ( CALL) C,  CELL GAP  alter ;
\ create the code field for colon words, DOES> and GOES>
: : code, ( 'code )   HERE  233 ( JMP ) C,  CELL GAP  alter ;
\ create the code field for data words.

: patch ( 'code cfa)   233 ( JMP ) OVER C!  alter ;
\ make 'code the new action of the cf.  Used by (;code).

SCREEN 41
\ Alignment, string and error primitives
\ : ALIGN   HERE 1 AND GAP ;            ( ALIGN)
\ force dictionary to the next even address.
\ : REALIGN ( a - a2)   DUP 1 AND + ;  ( ALIGN)
\ force address to the next even address.

: : (") ( - a l)   I> COUNT  2DUP +  ( REALIGN) >I ;
\ leave the address and length of an in-line string.

: : huh? ( w)   0= ABORT" ?" ;
\ error action of several words.

: ' ( - w)   BL WORD  DUP C@ huh?  FIND huh? ;

\ : I>   [COMPILE] R> ;  IMMEDIATE  ( ALIGN)
\ : >I   [COMPILE] >R ;  IMMEDIATE  ( ALIGN)

SCREEN 42


SCREEN 43
\ ------ Compiler --------------------------------
: COMPILE   I>  DUP CELL+ >I  @ , ;
\ compile the word that follows in the definition.

: header  \ create link and name fields.
   ( ALIGN)              'loc  @EXECUTE  ( extra fields )
   BL WORD  DUP C@ huh?  'warn @EXECUTE  ( redefinition?)
   HERE last !  HERE CURRENT @ DUP @ , ! ( link field)
   HERE OVER C@ 1+ CMOVE                 ( name field)
   HERE C@ DUP 128 OR C, GAP   HERE last CELL+ ! ;

SCREEN 44
\ Defining words
: CREATE ( - a)
   header  [ addr ] LITERAL code, ;

: VARIABLE ( - a)
   header  [ con ] LITERAL code,  THERE ,
   0 THERE ! ( courtesy )  CELL ALLOT ;

: CONSTANT ( - w)
   header  [ con ] LITERAL code,  , ;

SCREEN 45
\ DOES> and GOES>
: : (;code)   I> last CELL+ @ patch ;
\ the code field of (;code) is at  ' DOES> >BODY CELL+


: DOES>   COMPILE (;code)  [ (does) ] LITERAL nest, ; IMMEDIATE
\ eg  : KONST   CREATE  ,  DOES> @ ;

: GOES>  ( ROM)  [COMPILE] DOES>  COMPILE @ ;  IMMEDIATE
\ eg  : VALUE   VARIABLE   GOES> @ ;

SCREEN 46
\ Literals
: LITERAL ( - w)     COMPILE lit  , ;  IMMEDIATE
\ compile w as a literal.
: ['']     ( - w)  '  COMPILE tic  , ;  IMMEDIATE
\ compile-form of ' ("tick").

:  ASCII  ( - c)   BL WORD 1+ C@ ;  \ return value of next char.
: [ASCII] ( - c)                    \ compile value of next char.
   ASCII  [COMPILE] LITERAL ;  IMMEDIATE

: STRING ( c)  ( Feature)  \ string compiler, eg 32 STRING ABC
   parse  DUP C,  HERE OVER GAP  SWAP CMOVE  ( ALIGN) ;

: " ( - a u)    \ string literal, eg " cccc"
   COMPILE (")  [ASCII] " STRING ;  IMMEDIATE
: ."   [COMPILE] "  COMPILE TYPE ;  IMMEDIATE

SCREEN 47
\ Flow of control
: : ?bal   DUP bal @ < huh?  PICK @ 0= huh? ;
: : -bal   bal @ huh?  TRUE bal +!  DUP @ huh? ;

: BEGIN  HERE  1 bal +! ;                          IMMEDIATE

: IF     COMPILE ?branch  [COMPILE] BEGIN   0 , ;  IMMEDIATE
: THEN   0 ?bal  TRUE bal +!  HERE SWAP ! ;        IMMEDIATE
: ELSE   0 ?bal  COMPILE  branch  [COMPILE] BEGIN  0 ,
         SWAP   [COMPILE] THEN ;                   IMMEDIATE

: UNTIL  -bal  COMPILE ?branch  , ;                IMMEDIATE
: AGAIN  -bal  COMPILE  branch  , ;  ( Control?)   IMMEDIATE
: WHILE   bal @ huh?  [COMPILE] IF  SWAP ;         IMMEDIATE
: REPEAT  1 ?bal  [COMPILE] AGAIN [COMPILE] THEN ; IMMEDIATE

SCREEN 48
\ Definite loops
: DO   COMPILE (do)  [COMPILE] BEGIN ;   IMMEDIATE

: LEAVE  COMPILE >undo  COMPILE branch
   HERE  leaf @ , leaf ! ;                 IMMEDIATE

: : rake,   \ gathers leaf's.  Courtesy of Wil Baden.
   DUP ,  leaf @
   BEGIN  2DUP U< WHILE  DUP @ HERE ROT !  REPEAT
   leaf ! DROP ;
:  LOOP  -bal  COMPILE  (loop)  rake, ;  IMMEDIATE
: +LOOP  -bal  COMPILE (+loop)  rake, ;  IMMEDIATE

: UNDO   COMPILE >undo ;  IMMEDIATE

SCREEN 49
\ Colon definitions
: :   \ create a word and enter the compiling loop.
   CURRENT @ CONTEXT !
   header  [ colon ] LITERAL nest,
   last @ @ CONTEXT @ !  0 0 bal 2!  ] ;

: ;   \ terminate a definition.
   bal 2@ OR ABORT" Unbalanced"
   last @ CURRENT @ !
   COMPILE semi [COMPILE] [ ;  IMMEDIATE

SCREEN 50
\ Vocabularies
: FORTH   f CONTEXT ! ;

: DEFINITIONS   CONTEXT @ CURRENT ! ;
\ new definitions will be into the CURRENT vocabulary.

: VOCABULARY
\ when executed, a vocabulary becomes first in the search order.
   VARIABLE   HERE  f CELL+ ( ie vlink) DUP @ , !
   CELL GAP ( value for automatic initialization)
   GOES> CONTEXT ! ;

SCREEN 51
\ Misc. compiler support
: IMMEDIATE   last @ CELL+  DUP C@ BL ( ie 32) OR SWAP C! ;

: [COMPILE]    ' , ;  IMMEDIATE
\ force compilation of an otherwise immediate word.

:  (   [ASCII] ) parse  2DROP ;  IMMEDIATE   ( comments)
: .(   [ASCII] ) parse   TYPE ;  IMMEDIATE   \ messages.

: RECURSE   last CELL+ @ , ;  IMMEDIATE   \ self-reference.

SCREEN 52
( Hall of fame--  high-level)  EXIT
: M+ ( d n - d2)  ( Control?)  S>D D+ ;   \ add n to d.

: >< ( u - u2)  ( Control?)  DUP 255 AND  SWAP 256 * OR ;
\ reverse the bytes within a cell.

: WITHIN ( u n n2 - f)  ( Control?)  OVER - >R - R> U< ;
\ true if n <= u < n2  given circular comparison.

: ERASE ( a u)    0 FILL ;  ( CONTROLLED)
: BLANK ( a u)   BL FILL ;  ( CONTROLLED)

SCREEN 53
( Hall of fame--  low-level)
CODE M+ ( d n - d2)          ( Control?)  \ add n to d.
   BX AX XCHG  CWD   BX POP  CX POP  AX CX ADD  CX PUSH
   DX BX ADC   NEXT C;

CODE >< ( u - u2)  ( Control?)  BL BH XCHG   NEXT C;
\ reverse the bytes within a word.

: WITHIN ( u n n2 - f)  ( Control?)  OVER - >R - R> U< ;
\ true if n <= u < n2  given circular comparison.

: ERASE ( a u)    0 FILL ;  ( CONTROLLED)
```

(continued on page 104)

Listing One *(Listing continued, text begins on page 38.)*

```
: BLANK ( a u)   BL FILL ;  { CONTROLLED}

SCREEN 54
\ Byte move operators
: CMOVE> ( a a2 u)  { CONTROLLED}
\ move u bytes from a to a2, rightmost byte first.
   DUP DUP  >R  D+ R>  ?DUP
   IF  0 DO  1- SWAP 1-  TUCK C@ OVER C!  LOOP THEN 2DROP ;

: MOVE ( a a2 u)   \ move u bytes from a to a2 without overlap.
   >R  2DUP U< IF  R> CMOVE>  ELSE  R> CMOVE  THEN ;

: ROLL ( w[u] w[u-1]...w[0] u - w[u-1]...w[0] w[u])
\ rotate kth item to top of stack.  ( Delete?}
   DUP  BEGIN  ?DUP WHILE  ROT >R      1- REPEAT
        BEGIN  ?DUP WHILE  R> ROT ROT  1- REPEAT ;

SCREEN 55
( Double-number math-- high-level)  EXIT
: S>D ( n - d)   DUP 0< ;           \ extend n to d.
: D>S ( d - n)   DROP ;  { DOUBLE}  \ truncate d to n.
{ Require?}

: D- ( d d2 - d')   DNEGATE D+ ;  { DOUBLE}

: D2* ( d - d*2)   2DUP D+ ;
: D2/ ( d - d/2)   SWAP 2/ 32767 AND         { DOUBLE}
   OVER 1 AND IF  32768 OR  THEN  SWAP 2/ ;  { Require?}

SCREEN 56
( Double-number math-- low-level)
CODE S>D ( n - d)   \ extend n to d.
   BX AX XCHG  CWD  AX PUSH  BX DX XCHG   NEXT C;
CODE D>S ( d - n)   BX POP  NEXT C;  { Req?}  \ truncate d to n.
CODE D- ( d d2 - d3)   BX DX MOV  AX POP  BX POP  CX POP
   AX CX SUB  CX PUSH  DX BX SBB   NEXT C;   { DOUBLE}

CODE D2* ( d - d2)
   AX POP   AX SHL  BX RCL   AX PUSH   NEXT C;
CODE D2/ ( d - d2)  { DOUBLE}  { Require?}
   AX POP   BX SAR  AX RCR   AX PUSH   NEXT C;

SCREEN 57
\ More Double-number math
: D<  ( d d2 - f)
   ROT 2DUP = IF  2DROP U< EXIT THEN  2SWAP 2DROP > ;

: D0= ( d - f)          OR 0= ;    { DOUBLE}
: D=  ( d d2 - f)   D- OR 0= ;    { DOUBLE}

: DABS ( d - ud)   DUP 0< IF  DNEGATE  THEN ;  { Double?}

: DMAX ( d d2 - dmax)  { DOUBLE}
   2OVER 2OVER D<      IF 2SWAP THEN  2DROP ;
: DMIN ( d d2 - dmin)  { DOUBLE}
   2OVER 2OVER D< NOT IF 2SWAP THEN  2DROP ;

SCREEN 58
\ Double-number operators
: 2CONSTANT ( - w)   CREATE  , ,  DOES> 2@ ;
\ create a double constant.  { DOUBLE}
: 2VARIABLE ( - a)   VARIABLE  0 THERE ! CELL ALLOT ;
\ create a double variable.  { DOUBLE}

: D@ ( a - d)   2@ ;  { DOUBLE}
: D! ( d a )    2! ;  { DOUBLE}

: DLITERAL ( d ) ( - d)  { Double?}  \ compile d as a literal.
   SWAP [COMPILE] LITERAL [COMPILE] LITERAL ; IMMEDIATE

: D.R ( d n)  { DOUBLE}
\ print d right-justified in field of width n.
   >R  TUCK  DABS  <#  #S ROT SIGN  #>
   R>  OVER - 0 MAX SPACES  TYPE ;

SCREEN 59
( Mixed-precision multiply and divide-- high-level)  EXIT
: M* ( n n2 - d)  { Control?}
\ signed mixed-precision multiply.
   2DUP XOR >R  ABS SWAP ABS UM*  R> 0< IF NEGATE THEN ;

: M/MOD ( d n - rem quot)  { Control?}
\ signed rounded-down mixed-precision divide.
   2DUP XOR >R  OVER >R  ABS >R DABS R> UM/MOD
   SWAP R> 0< IF  NEGATE  THEN
   SWAP R> 0< IF  NEGATE  THEN ;

SCREEN 60
( Mixed-precision multiply and divide-- low-level)
CODE M* ( n n2 - d)  { Control?}
\ signed mixed-precision multiply.
   BX AX XCHG  DX POP   DX IMUL   AX PUSH  DX BX MOV   NEXT C;

CODE M/MOD ( d n - rem quot)  { Control?}  DX POP  AX POP
\ signed rounded-down mixed-precision divide.
   BX BX OR  5 L# JZ  ( divide by zero?)
   BX IDIV        AX BX MOV  DX PUSH   NEXT
 5 L: AX DX MOV  0 # BX MOV  DX PUSH   NEXT C;

SCREEN 61
( Mixed-precision multiply and divide-- floored)  EXIT
CODE M* ( n n2 - d)  { Control?}
\ signed mixed-precision multiply.
   BX AX XCHG  DX POP   DX IMUL   AX PUSH  DX BX MOV   NEXT C;

: M/MOD ( d n - rem quot)  { Control?}
\ signed floored mixed-precision divide.
   DUP >R  2DUP XOR >R  DUP >R  ABS >R DABS R> UM/MOD
   SWAP R> 0< IF  NEGATE  THEN
   SWAP R> 0< IF  NEGATE  OVER IF  R@ ROT -  SWAP 1-  THEN THEN
   R> DROP ;

SCREEN 62
\ Multiply and divide
: /MOD ( n n2 - n3 n4)   >R DUP 0< R> M/MOD ;

: /   ( n n2 - n3)   /MOD  NIP ;
: MOD ( n n2 - n3)   /MOD  DROP ;

\ Intermediate product is 32 bits:
: */MOD ( n n2 n3 - n4 n5)   >R M* R> M/MOD ;
: */    ( n n2 n3 - n4)      >R M* R> M/MOD  NIP ;

CODE * ( n n2 - n3)   AX POP  BX IMUL   AX BX MOV   NEXT C;

EXIT
: * ( n n2 - n3)   UM* DROP ;

SCREEN 63
\ Number conversion operator
: : val? ( a u - d 2 , n 1 , 0)
\ string to number conversion primitive.  True if d is valid.
\ Returns d if number ends in final '.' and sets dpl = 0
\ Returns n if no punctuation present   and sets dpl = 0<
   [ #Jot 1- ] LITERAL MIN  PAD 1- OVER -  TUCK >R  CMOVE

   BL PAD 1-  DUP dpl ! C!  0 0 R>
   DUP C@ [ASCII] - = DUP >R - 1-
   BEGIN  CONVERT  DUP C@  DUP [ASCII] : =
     SWAP [ASCII] , [ASCII] / 1+ WITHIN  OR
   WHILE  DUP dpl !  REPEAT  R> SWAP >R IF  DNEGATE  THEN
   PAD 1- dpl @ - 1- dpl !   R> PAD 1- = ( valid?)
   IF  dpl @ 0< IF DROP 1 ELSE 2 THEN  ELSE  2DROP 0  THEN ;

: VAL? ( a u - d 2 , n 1 , 0)  { Feature}  'val? @EXECUTE ;

SCREEN 64
\ Interpreter proper
: : val, ( ... w )
\ compiles the top w stack items as numeric literals.
   DUP BEGIN  ROT >R                1- ?DUP 0= UNTIL
       BEGIN  R> [COMPILE] LITERAL  1- ?DUP 0= UNTIL ;

: interpret  { Feature}  \ the text compiler loop.
   BEGIN  BL WORD  FIND  ?DUP
     IF    STATE @ =  ( Imm?) IF  ,  ELSE EXECUTE  THEN
     ELSE  COUNT VAL?  DUP huh?
           STATE @ IF  val,  ELSE  DROP  THEN
     THEN
   AGAIN ;

SCREEN 65
\ QUIT support
: EVALUATE ( a u)   \ evaluate a string.
   #TIB 2@ 2>R  #TIB 2!  BLK 2@ 2>R  0 0 BLK 2!  interpret
   2R> BLK 2!  2R> #TIB 2! ;

: EXPECT ( a +n)   'expect @EXECUTE ;

: QUERY  { CONTROLLED}
\ fill TIB from next line of input stream.
   0 0 BLK 2!  TIB 80 EXPECT  SPAN @ #TIB ! ;

: ok?   \ status check.
   D0 @ [ #Safe ] LITERAL -  HERE U< ABORT" No Room"
   DEPTH 0< ABORT" Stack?" ;

: OK?  { Feature}  ok?  STATE @ 0= IF ." ok" THEN ;

SCREEN 66
\ QUIT and ABORT
: QUIT   \ default main program.
   RESET  BEGIN  CR QUERY  SPACE interpret  OK?  AGAIN ;

: GRIPE ( a u)  { Feature}  \ default error handler.
   BLK @ IF  BLK 2@ scr 2!  THEN
   THERE COUNT TYPE SPACE  ( msg ) TYPE ;

: ABORT   BEGIN  PRESET QUIT  GRIPE  AGAIN ;
\ default main program and error handler, courtesy Wil Baden.

: ABORT"   \ compile error handler and message.
   [COMPILE] IF  [COMPILE] "  COMPILE err  [COMPILE] THEN ;
  IMMEDIATE

SCREEN 67
( Debug-- EXIT when done)
: .S  { Control?}  \ display the data stack.
   DEPTH 0 MAX  ?DUP
   CR  IF 0 DO  DEPTH I - 1- PICK  .  LOOP  THEN  ." <-Top " ;

: DUMP ( a u)  { RESERVED}  \ simple dump.
   SPACE 0 DO  DUP 7 AND 0= IF  SPACE  THEN  DUP C@ .  1+ LOOP
   DROP ;

: ? ( a)   @ . ;  { Control?}

: WORDS  { Control?}  \ simple word list.
   CONTEXT @
   BEGIN @ ?DUP
   WHILE  DUP CELL+ COUNT 31 AND TYPE SPACE  REPEAT ;

SCREEN 68
\ FORGET support
: : clip ( a 'lfa)   \ unlink words below the given address.
   BEGIN   DUP @
   WHILE  2DUP @  SWAP U< NOT ( ie U<= )
     IF    DUP @ @ OVER ! ( unlinks it )  ELSE  @  THEN
   REPEAT  2DROP ;

: crop ( lfa)
\ crop dictionary to the given link address.
```

```
   f CELL+ ( ie vlink)  2DUP clip
   BEGIN  @ ?DUP WHILE  2DUP CELL - @ ( ie >RAM) clip
   REPEAT  FORTH DEFINITIONS  DUP CURRENT @ clip  h ! ;
```

SCREEN 69

```
\ FORGET and variations
: GUARD   h H0 3 CELLS CMOVE  THERE T0 ! ;  { Feature}
: EMPTY   H0 h 3 CELLS CMOVE  T0 @  r  ! ;  { Feature}

: >link ( cfa - lfa)
   BEGIN  1- DUP C@ 128 AND UNTIL  CELL - ;

: FORGET   \ forget words from the following <name>.
   CURRENT @ CONTEXT !   ' >link
   DUP HERE H0 @ WITHIN ABORT" Can't"  crop ;
{ FORGET cannot recover RAM and so is not ROMable.}
{ Delete?}
```

SCREEN 70

SCREEN 71

```
\ ------ Device drivers ---------------------------
HEX
: CODE (type) ( a u)   BX CX MOV  DX POP   1 # BX MOV
   40 # AH MOV  21 INT   BX POP   'pause ) JMP C;

: CODE KDOS ( - key -1 , ? 0)
\ check for key pressed.
\ Special keys are returned in high byte with low byte zeroed.
   BX PUSH   FF # DL MOV   6 # AH MOV   21 INT
   0 # BX MOV  2 L# JE      AH AH SUB   ( special key?)
   AL AL OR    1 L# JNZ    7 # AH MOV   21 INT
   AH AH SUB   AL AH XCHG
 1 L: TRUE # BX MOV   2 L: AX PUSH   'pause ) JMP C;
```

SCREEN 72

```
\ KEY and EMIT actions
  13 EQU #EOL ( end-of-line)   10 EQU #LF  ( line-feed)
HERE EQU $Eol  #EOL C, #LF C,   2 EQU #Eol

: : (cr)   $Eol #Eol (type) ;

: : (key?) ( - f)   \ true if key pressed since last KEY.
   key' @ 0= IF  KDOS  key' 2!  THEN  key' @ ;

: KEY ( - n)   BEGIN  (key?) UNTIL  key' CELL+ @  0 key' ! ;
: EMIT ( b)   hld C!  hld 1 TYPE ;
```

SCREEN 73

```
\ EXPECT action
08   EQU #BSP ( backspace)    127 EQU #DEL ( delete)
27   EQU #ESC ( escape)
HERE EQU $Bsp ( * ) 3 C, #BSP C, BL C, #BSP C,

: : expect ( a +n)   >R  0  ( a o)
\ read upto +n chars into address; stop at #EOL or #ESC
   BEGIN  DUP R@ <
   WHILE  KEY 127 ( 7-bit ASCII) AND
     DUP #BSP =  OVER #DEL = OR
     IF   DROP  DUP IF  1-  $Bsp COUNT TYPE  THEN
     ELSE  DUP #EOL = OVER #ESC = OR
       IF  DROP  SPAN !  R> 2DROP  EXIT  THEN
       ( otherwise) BL MAX >R  2DUP +  R> OVER C!  1 TYPE  1+
     THEN
   REPEAT  SPAN !  R> 2DROP ;
```

SCREEN 74

```
\ Dumb terminal actions
: : (keys) ( a +n)   >R  0  ( a o)
\ read upto +n chars into address without echo; stop at #EOL
   BEGIN  DUP R@ <
   WHILE  KEY  DUP #EOL =
     IF  R> 2DROP  DUP >R  ( early out)
     ELSE  BL MAX >R  2DUP +  R> SWAP C!  1+  THEN
   REPEAT  SPAN !  R> 2DROP ;

: : (mark) ( a n)    ." ^"  TYPE ;
: : (page)   25 0 DO  CR  LOOP ;
: : (tab)   ( n n2)    CR  DROP SPACES ;
```

SCREEN 75

```
\ Initialize automatic variables
HERE EQU RAMs
] nope expect source nope nope  val? [
( key' ) 0 , 0 ,
HERE RAMs - EQU #RAMs
```

SCREEN 76

SCREEN 77

```
\ ------ Initialization ---------------------------
D0 CONSTANT parms   \ System parameter table.

CREATE glass        \ Simple transfer table.
] (type) (cr)  (keys) (key?)  (mark) (page)  (tab) nope [

: READY  ." Ready" ;  { Feature}  \ Initialize application.
: BYE    0 EXECUTE ;  { Feature}  \ Shut down  application.
```

SCREEN 78

```
\ Initialization-- high-level
160 CONSTANT VERSION  { Feature}  \ ZEN 1.60

: : vocabs   \ initialize vocabularies.
   f CELL+ ( ie vlink)
```

(continued on page 106)

CIRCLE NO. 69 ON READER SERVICE CARD

Listing One *(Listing continued, text begins on page 38.)*

```
   BEGIN  @ ?DUP
   WHILE  DUP CELL+ @ OVER CELL - @ ( ie >RAM) !  REPEAT ;

: cold   \ high-level coldstart initialization.
   TRUE ( wake) entry entry 2!  TO 2@ r 2!   glass x !
   RAMs 'pause #RAMs CMOVE
   EMPTY  vocabs  PRESET  FORTH DEFINITIONS  DECIMAL
   " READY" EVALUATE   ABORT ;
\ If all definitions are headerless, substitute:  READY  ABORT ;

SCREEN 79
\ Initialization-- low-level
HEX  HERE ( *) ," No Room $"

: CODE Coldstart   \ low-level initialization.
   1000 # BX MOV   4A # AH MOV   21 INT  ( enough room?)
 1 L# JNC  ( No:)
   ( *) 1+ # DX MOV   9 # AH MOV   21 INT   0 # JMP  ( Bye)
 1 L: #SP0 # SP MOV   #RP0 # BP MOV   BP u ) MOV
   ' cold >BODY # SI MOV  ( I register)   NEXT C;

HERE ( * ) Power ORG   ASSEMBLER  ' Coldstart # JMP C;
     ( * )       ORG


SCREEN 80

SCREEN 81
\ ------ FILE extension ---------------------------
#Used USER IO-RESULT  DROP

26  EQU #EOF  \ control-Z marks the end of older text files.

128 EQU buff  \ MS-DOS command tail and default fcb buffer.
192 EQU name  \ RENAME-FILE takes two names.

256  buff - EQU #buff   \ size of buffer in bytes.
name buff - EQU #name   \ size of name in bytes plus zero.

: >fname ( a u - a2)   \ convert string to ASCIIZ file name.
   buff  2DUP 2>R  SWAP MOVE  R@  0 2R> + C! ;

SCREEN 82
\ MS-DOS interface
HEX
CODE fdos ( DX CX handle function# - AX)
\ generic call to MS-DOS
   BX AX MOV  BX POP  CX POP  DX POP   21 INT
LABEL return   AX BX MOV   1 L# JB   AX AX SUB   2 L# JZ
 1 L: BX BX SUB ( non-zero retcode forces zero result)
 2 L: u ) DI MOV   AX IO-RESULT entry - [DI] MOV   NEXT C;

: CODE rename ( a a2 function# - AX)
   BX AX MOV  DI POP  DX POP   21 INT   return JU C;

: CODE seek ( DX CX handle function# - AX DX)
   BX AX MOV  BX POP  CX POP  DX POP   21 INT
   DX PUSH   return JU C;

SCREEN 83
\ 5 file primitives
HEX
: OPEN-FILE   ( a u - w)   >fname  0 0 3D02 fdos ;
: CREATE-FILE ( a u - w)   >fname  0 0 3C00 fdos ;

: DELETE-FILE ( a u)   >fname  0 0 4100 fdos  DROP ;
: CLOSE-FILE  ( w)     0 0 ROT     3E00 fdos  DROP ;

: RENAME-FILE ( a u a2 u2)
   >fname  name #name CMOVE>
   >fname  name  5600 rename  DROP ;

SCREEN 84
\ Read, write and seek bytes
HEX
\ Read or write u bytes to or from address a to file w.
: READ-FILE  ( a u w - u2)   3F00 fdos ;
: WRITE-FILE ( a u w - u2)   4000 fdos ;

: SEEK-FILE ( doff n w - dpos)   \ add an offset to file w.
\ n neg: to start; n pos: to end; n zero: to current.
   SWAP DUP IF  0< CELLS 1+  THEN  4201 + seek ;

\ Return file position or size.
: FILEPOS  ( w - d)   >R  0 0 0 R> SEEK-FILE ;
: FILESIZE ( w - d)   >R  0 0 1 R> SEEK-FILE ;

SCREEN 85
\ Read and write lines of text
: WRITE-CR ( w)   $Eol #Eol ROT WRITE-FILE  DROP ;

: READ-LINE ( a u w - 0 0 | u2 t)
{ Greater performance will result if the end-of-line sequence  }
{ is read into the address and the size u adjusted accordingly.}
   >R  buff OVER 1+ #buff MIN R@ READ-FILE  ( a u u2)
   DUP 0= IF  R> 2DROP 2DROP  0 0 EXIT THEN ( end of file)
   buff OVER #EOL SCAN  NIP ( a u u2 u3)
   ?DUP IF   #Eol OVER - >R -
         ELSE  2DUP U< >R  THEN  MIN R> ( a u4 #seek)
   ?DUP IF  S>D 0 R@ SEEK-FILE 2DROP  THEN
   buff OVER #EOF SCAN  NIP -   ( remove if no control-Zs)
   R> DROP ( a u4) >R  buff SWAP R@ CMOVE>  R> TRUE ;

SCREEN 86
\ Load and save files
: GO ( a u)  { Feature}  \ evaluate the KERNEL.SRC file.
   " KERNEL.SRC" OPEN-FILE DUP huh?  ( w) >R
   BEGIN  buff  DUP 64 R@ READ-LINE
   WHILE  EVALUATE  REPEAT  2DROP   R> CLOSE-FILE ;
: SAVE-FILE ( a u)  { Feature}  \ save the dictionary by name.
   CREATE-FILE DUP huh?  ( w) >R
   'pause RAMs #RAMs CMOVE   GUARD   f CELL+ ( ie vlink)
   BEGIN  @ ?DUP ( save vocabularies)
   WHILE  DUP CELL - @ ( ie >RAM) @ OVER CELL+ !  REPEAT
   256  HERE OVER - R@ WRITE-FILE  DROP   R> CLOSE-FILE ;

SCREEN 87
\ BLOCK word set
VARIABLE system    VARIABLE block#    VARIABLE update
VARIABLE buffer  1024 CELL - ALLOT

: seek-block ( u w - a n w)
   >R  1024 UM* TRUE  R@ SEEK-FILE 2DROP  buffer 1024 R> ;

: SAVE-BUFFERS  { BLOCK}  system @ 0= ABORT" No File"
   system 2@ seek-block WRITE-FILE DROP ;

: BUFFER ( u - a)  { BLOCK}  >R  block# 2@  R@ - AND
   IF  SAVE-BUFFERS  THEN   0 R> block# 2!  buffer ;

: BLOCK  ( u - a)  { BLOCK}
   DUP block# @ = IF  DROP buffer  EXIT THEN
   BUFFER >R  system 2@ seek-block READ-FILE DROP  R> ;

SCREEN 88
\ BLOCK support
: EMPTY-BUFFERS  { CONTROLLED}  0 TRUE block# 2! ;

HEX
: UPDATE  { BLOCK}  TRUE update ! ;
: FLUSH   { BLOCK}
   SAVE-BUFFERS  0 0 system @ 4500 fdos  CLOSE-FILE ;

: LOAD ( u)  { BLOCK}
   BLK 2@ 2>R  0 SWAP BLK 2!  interpret  2R> BLK 2! ;

: block   BLK @ ?DUP IF  BLOCK 1024  ELSE  #TIB 2@  THEN ;

\ Use this definition if the BLOCK word set is compiled:
: READY   ." Ready!"    ['] block 'source !
   " NEW.SCR" OPEN-FILE  DUP huh?  system !  EMPTY-BUFFERS ;
```

End Listing

Listing One *(Text begins on page 48.)*

```
/* Include file for macros replacement of error function. */

#define error(m)  _error(__FILE__, __LINE__)

extern  void      _error(char *, int);
```

End Listing One

Listing Two

```
/* Definition of error function.  */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "error.h"

static int get_err_msg(char *, char *, char *);

void
_error(char *fname, int lno)
{
    char *period;  /* Pointer for file extension */
    char lstr[6];  /* Temp string for file line */
    char key[15];  /* Message key (file:line) */
    char msg[75];  /* Error message from file */

    /* Strip out file extension from file name */
    if ((period = strchr(fname, '.')) != NULL)
        *period = '\0';

    /* Create file:linenumber key */
    sprintf(key, "%s:%s", fname, itoa(lno, lstr,
       10));

    /* Look up error message from file */
    if (!get_err_msg(fname, key, msg))
        /* Not found in file, just print out
         * file:lineno.
         */
        printf("Error [%s]", key);
    else
        printf("%s [%s]", msg, key);
    return;
} /* error() */

static int
get_err_msg(char *fname, char *key, char *msg)
{
    FILE *fp;      /* Error message file pointer */
    char msg_key[14];/* Key for current line */

    /* Open error file */
    if (!(fp = fopen("error.dat", "r")))
        return FALSE;

    /* Scan file for message */
    while (!feof(fp)) {
        fscanf(fp, " %s ", msg_key);
        fgets(msg, 75, fp);
        if (!strcmpi(key, msg_key))
            break;
    }
    fclose(fp);

    /* Return false if message not found */
    if (feof(fp))
        return FALSE;

    /* Remove CR from message string */
    msg[strlen(msg)-1] = '\0';
    return TRUE;
} /* get_err_msg() */
```

End Listing Two

Listing Three

```
BEGIN {
    FS = "\"";
    printf("") > "error.txt";
    printf("") > "error.dat";
}
/ *error *\(/ {
    printf("%s :\n", $2) >> "error.txt";
    ind = substr(FILENAME, 0, index(FILENAME, ".")
      - 1);
    printf("%s:%d\t%s\n", ind, NR, $2) >>
      "error.dat";
    getline();
    i = index($0, "/* ");
    if (i) i++;
    while (i) {
        printf("\t%s\n", substr($0, i + 2)) >>
          "error.txt";
        getline();
        if (index($0, "*/")) break;
        i = index($0, "* ");
    }
}
```

End Listings

Listing One *(Text begins on page 52.)*

```
/*****************************************************************/
/*      Simple S-CODER file encryptor/decryptor                  */
/*****************************************************************/

#include <stdio.h>
#include <string.h>
#include <assert.h>

#define BUF_SIZ 32768

extern char *cryptext;
extern int  crypt_length;
void crypt(char *);

main(int argc, char *argv[])
{
        unsigned i, n;
        char *buf, *p;
        FILE *infile, *outfile;

        if (4 > argc)
        {
                puts("\aUsage: CRYPT input_file output_file key");
                abort();
        }
        assert(buf = (char *)malloc(BUF_SIZ));
        assert(infile  = fopen(argv[1], "rb"));
        assert(outfile = fopen(argv[2], "wb"));
        cryptext = argv[3];
        crypt_length = strlen(cryptext);
        while (n = fread(buf, 1, BUF_SIZ, infile))
        {
                p = buf;
                for (i = 0; i < n; ++i)
                        crypt(p++);
                fwrite(buf, 1, n, outfile);
        }
        fclose(infile);
        fclose(outfile);
        exit(0);
}
```

End Listing One

Listing Two

```
/*****************************************************************/
/*      Simple S-CODER stream encryptor/decryptor                */
/*****************************************************************/

#include <stdio.h>
#include <string.h>
#include <assert.h>

extern char *cryptext;
extern int  crypt_length;
void crypt(char *);

main(int argc, char *argv[])
{
        char cch;
        int ich;
        FILE *infile;
        void setraw(void);

        if (2 > argc)
        {
                puts("\aUsage: SCRYPT key");
                puts("encrypts stdin to stdout");
                abort();
        }
        cryptext = argv[1];
        crypt_length = strlen(cryptext);
        setraw();  /* NOTE: setraw() will be compiler-dependent. It is used
                      to set stdin and stdout to raw binary mode. This is
                      necessary to avoid CR/LF translation and to avoid
                      sensing 0x1a as EOF during decryption. */
        while (EOF != (ich = getchar()))
        {
                cch = (char)ich;
                crypt(&cch);
                fputc(cch, stdout);
        }
        exit(0);
}
```

End Listing Two

Listing Three

```
/*****************************************************************/
/*      Zortech C routine to set stin and stdout to binary mode */
/*****************************************************************/

#include <stdio.h>

extern FILE _iob[_NFILE];

void setraw(void)
{
        _iob[0]._flag &= ~_IOTRAN;
        _iob[1]._flag &= ~_IOTRAN;
}
```

End Listing Three

Listing Four

```
/*****************************************************************/
/*      Enhanced security S-CODER file encryptor/decryptor       */
/*****************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#define MIN_KEYL 6

extern char *cryptext;
extern int  crypt_length;
void crypt(char *);
int cryptqual(void);
long fsize;
union {                         /* Transposition cipher buffer  */
        char in[16384];
        char out[128][128];
} buf;
FILE *infile, *outfile;

main(int argc, char *argv[])
{
        void encrypt(void);
        void decrypt(void);

        if (5 > argc || NULL == strchr("EeDd", argv[1][0]))
        {
                puts("\aUsage: HI-CRYPT { E | D } input_file output_file key");
                puts("where: E = Encrypt");
                puts("       D = Decrypt");
                abort();
        }
        assert(infile  = fopen(argv[2], "rb"));
        assert(outfile = fopen(argv[3], "wb"));
        cryptext = argv[4];
        crypt_length = strlen(cryptext);
        if (cryptqual())
        {
                puts("\aHI-CRYPT: Key is not sufficiently complex");
                abort();
        }
        if (strchr("Ee", argv[1][0]))
                encrypt();
        else    decrypt();
        fclose(infile);
        fclose(outfile);
        exit(0);
}
int cryptqual(void)
{
        int i, j = 0;
        static char found[MIN_KEYL + 1];   /* Statics initialized to zeros */

        if (MIN_KEYL > crypt_length)
                return -1;
        for (i = 0; i < crypt_length; ++i)
        {
                if (strchr(found, cryptext[i]))
                        continue;
                found[j++] = cryptext[i];
                if ((MIN_KEYL - 1) < j)
                        return 0;
        }
        return -1;
}
void encrypt(void)
{
        unsigned i, j, n;

        fseek(infile, 0L, SEEK_END);
        fsize = ftell(infile);                          /* Save size    */
        rewind(infile);
        fwrite(&fsize, sizeof(long), 1, outfile);
        srand((unsigned)fsize);
        crypt_ptr = fsize % crypt_length;
        while (n = fread(buf.in, 1, 16384, infile))
        {
                while (16384 > n)
                        buf.in[n++] = rand();
                for (i = 0; i < 128; ++i)
                        for (j = 0; j < 128; ++j)
                                crypt(&buf.out[j][i]);
                fwrite(buf.in, 1, 16384, outfile);
        }
}
void decrypt(void)
{
        unsigned i, j, n;
        fread(&fsize, sizeof(long), 1, infile);
        crypt_ptr = fsize % crypt_length;
        while (n = fread(buf.in, 1, 16384, infile))     /* Read size    */
        {
                for (i = 0; i < 128; ++i)
                        for (j = 0; j < 128; ++j)
                                crypt(&buf.out[j][i]);
                if (16384 <= fsize)
                        fwrite(buf.in, 1, 16384, outfile);
                else    fwrite(buf.in, 1, fsize, outfile);
                fsize -= n;
        }
}
```

End Listing Fo

Listing Five

```
/*****************************************************************/
/*      Collect file statistics                                  */
/*****************************************************************/

#include <stdio.h>
#include <math.h>
#include <assert.h>

main(int argc, char *argv[])
{
        int i, ch, hist = 0;
        long n = 0L;
        double mean = 0., stdev = 0., ftmp;
        static unsigned bins[256];
        FILE *infile;

        assert(infile = fopen(argv[1], "rb"));
        while (!feof(infile))
        {
                if (EOF == (ch = fgetc(infile)))
                        break;
                bins[ch] += 1;
                ++n;
        }
        fclose(infile);
        for (i = 0; i < 256; ++i)
        {
                mean += (double)(bins[i]);
                if (bins[i])
                        ++hist;
        }
        mean /= 256.;
        for (i = 0; i < 256; ++i)
        {
                ftmp = (double)(bins[i]) - mean;
                stdev += (ftmp * ftmp);
        }
        ftmp  = stdev / 255.;
        stdev = sqrt(ftmp);
        printf("%ld Characters were read from %s\n"
                "There are an average of %f occurances of each character\n"
                "%d Characters out of 256 possible were used\n"
                "The standard deviation is %f\n"
                "The coefficient of variation is %f%%\n",
                n, argv[1], mean, hist, stdev, (100. * stdev) / mean);
```

End Listings

Listing One *(Listing continued, text begins on page 64.)*

```
inline Complex polar(double Radius, double Theta)
    {
    Complex Result;
    Result.Real = Radius * cos(Theta);
    Result.Imag = Radius * sin(Theta);
    return Result;
    }
inline Complex conj(const Complex & C)
    {
    Complex Result;
    Result.Real =  C.Real;
    Result.Imag = -C.Imag;
    return Result;
    }
// trigonometric methods
inline Complex cos(const Complex & C)
    {
    Complex Result;
    Result.Real =  cos(C.Real) * cosh(C.Imag);
    Result.Imag = -sin(C.Real) * sinh(C.Imag);
    return Result;
    }
inline Complex sin(const Complex & C)
    {
    Complex Result;
    Result.Real = sin(C.Real) * cosh(C.Imag);
    Result.Imag = cos(C.Real) * sinh(C.Imag);
    return Result;
    }
inline Complex tan(const Complex & C)
    {
    Complex Result;
    Result = sin(C) / cos(C);
    return Result;
    }
inline Complex cosh(const Complex & C)
    {
    Complex Result;
    Result = cos(C.Imag) * cosh(C.Real);
    Result = sin(C.Imag) * sinh(C.Real);
    return Result;
    }
inline Complex sinh(const Complex & C)
    {
    Complex Result;
    Result.Real = cos(C.Imag) * sinh(C.Real);
    Result.Imag = sin(C.Imag) * cosh(C.Real);
    return Result;
    }
inline Complex tanh(const Complex & C)
    {
    Complex Result;
    Result = sinh(C) / cosh(C);
    return Result;
    }
// logarithmic methods
inline Complex exp(const Complex & C)
    {
    Complex Result;
    double X = exp(C.Real);
    Result.Real = X * cos(C.Imag);
    Result.Imag = X * sin(C.Imag);
    return Result;
    }
inline Complex log(const Complex & C)
    {
    Complex Result;
    double Hypot = abs(C);
    if (Hypot > 0.0)
        {
        Result.Real = log(Hypot);
        Result.Imag = atan2(C.Imag, C.Real);
        }
    else
        Complex::ErrorHandler();
    return Result;
    }
#endif // __Complex_HPP
```

End Listing One

Listing Two

```
//  Module:     Complex
//  Version:    2.00    28-Oct-1989
//  Language:   C++ 2.0;  Environ:    Any;  Compilers:  Zortech C++ 2.01
//  Purpose:    Provides the class "Complex" for C++ programs. The majority
//              of the class is implemented inline for efficiency. Only
//              the division, power, and i/o methods are actual functions.
//  Written by: Scott Robert Ladd, 705 West Virginia, Gunnison CO 81230
//              BBS (303)641-6438; FidoNet 1:104/708

#include "math.h"
#include "stdlib.h"
#include "Complex.hpp"
#include "stream.hpp"

static void DefaultHandler();
void (* Complex::ErrorHandler)() = DefaultHandler;
static void DefaultHandler()
    {
    cout << "\aERROR in complex object: DIVIDE BY ZERO\n";
    exit(1);
    }
// division methods
Complex operator / (const Complex & C1, const Complex & C2)
    {
    Complex Result;
    double Den;
    Den = norm(C2);
    if (Den != 0.0)
        {
        Result.Real = (C1.Real * C2.Real + C1.Imag * C2.Imag) / Den;
        Result.Imag = (C1.Imag * C2.Real - C1.Real * C2.Imag) / Den;
        }
    else
        Complex::ErrorHandler();
    return Result;
    }
Complex Complex::operator /= (const Complex & C)
    {
    double Den, OldReal;
    Den = norm(C);
    if (Den != 0.0)
        {
        OldReal = Real;
        Real = (Real * C.Real + Imag * C.Imag) / Den;
        Imag = (Imag * C.Real - OldReal * C.Imag) / Den;
        }
    else
        Complex::ErrorHandler();
    return *this;
    }
// "power" methods
Complex pow(const Complex & C, const Complex & Power)
    {
    Complex Result;
    if (Power.Real == 0.0 && Power.Imag == 0.0)
        {
        Result.Real = 1.0;
        Result.Imag = 0.0;
        }
    else
        {
        if (C.Real != 0.0 || C.Imag != 0.0)
            Result = exp(log(C) * Power);
        else
            Complex::ErrorHandler();
        }
    return Result;
    }
Complex sqrt(const Complex & C)
    {
```

(continued on page 117)

Listing Two *(Listing continued, text begins on page 64.)*

```
    Complex Result;
    double r, i, ratio, w;
    if (C.Real != 0.0 || C.Imag != 0.0)
        {
        r = C.Real < 0.0 ? -C.Real : C.Real;
        i = C.Imag < 0.0 ? -C.Imag : C.Imag;
        if (r > i)
            {
            ratio = i / r;
            w = sqrt(r) * sqrt(0.5 * (1.0 + sqrt(1.0 + ratio * ratio)));
            }
        else
            {
            ratio = r / i;
            w = sqrt(i) * sqrt(0.5 * (ratio + sqrt(1.0 + ratio * ratio)));
            }
        if (C.Real > 0)
            {
            Result.Real = w;
            Result.Imag = C.Imag / (2.0 * w);
            }
        else
            {
            Result.Imag = (C.Imag > 0.0) ? w : - w;
            Result.Real = C.Imag / (2.0 * Result.Imag);
            }
        }
    else
        Complex::ErrorHandler();
    return Result;
    }
// output method
ostream & operator << (ostream & Output, const Complex & C)
    {
    Output << form("(%+lg%+lgi)",C.Real,C.Imag);
    return Output;
    }
istream & operator >> (istream & Input, Complex & C)
    {
    char Ch;
    C.Real = 0.0;
    C.Imag = 0.0;
    Input >> Ch;
    if (Ch == '(')
        {
        Input >> C.Real >> Ch;
        if (Ch == ',')
            Input >> C.Imag >> Ch;
        if (Ch != ')')
            Input.clear(_bad);
        }
    else
        {
        Input.putback(Ch);
        Input >> C.Real;
        }
    return Input;
    }
```

End Listing Two

Listing Three

```
//  Program:    Biomorph    (Generate non-standard fractals)
//  Version:    1.01        31-Oct-1989
//  Language:   C++ 2.0;  Environ:  Any;  Compilers:  Zortech C++ 2.01
//  Purpose:    Generates fractals based on complex number formula iterations
//  Written by: Scott Robert Ladd, 705 West Virginia, Gunnison CO 81230
//              BBS (303)641-6438;  FidoNet 1:104/708

#include "conio.h"
#include "zg_lwlvl.h"
#include "complex.hpp"

Complex C, Z, Power;
double Range, Xinc, Yinc, Xmax, Ymax, Xorig, Yorig;
int X, Y, I, Iterations, Species;

void GetParams();
int  main();

void GetParams()
    {
    cout << "Biomorph 1.01 -- a complex-plane fractal generator\n";
    cout << "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n";

    cout << "This program generates these species of biomorphs...\n";
    cout << "  Species 0: Z^X + C          Species 1: sin(Z) + exp(Z) + C\n";
    cout << "  Species 2: Z^Z + Z^X + C    Species 3: sin(Z) + Z^X + C\n\n";
    do  {
        cout << "What species of biomorph do you want (0..3)? ";
        cin  >> Species;
        }
```

(continued on page 119)

Listing Three *(Listing continued, text begins on page 64.)*

```
    while ((Species < 0) || (Species > 3));
    cout << "\we need one or two complex numbers. These can be entered\n";
    cout << "in the following formats (where 'f' indicates a floating-point\n";
    cout << "value:\n";
    cout << "   f -or- (f) (just a real number)\n";
    cout << "   (f,f)      (entering both the real and imaginary parts)\n\n";
    if (Species != 1)
        {
        cout << "Enter the complex power applied to Z: ";
        cin  >> Power;
        }
    cout << "Enter the complex constant C: ";
    cin  >> C;
    cout << "\nNext two numbers are floating point values representing the\n";
    cout << "origin point on the complex plane of the area being viewed.\n\n";
    cout << "Enter the X location of the center of the picture: ";
    cin  >> Xorig;
    cout << "Enter the Y location of the center of the picture: ";
    cin  >> Yorig;
    cout << "\nNext number represents the distance the graph extends away\n";
    cout << "from the above origin.\n\n";
    cout << "Enter the range of the graph: ";
    cin  >> Range;
    cout << "\nFinally, how many iterations should the program perform? ";
    cin  >> Iterations;
    }

int main()
    {
    GetParams();
    if (ZG_Init()) return 1;
    if (ZG_SetMode(ZG_MOD_BESTRES)) return 2;
    Ymax  = ZG_VideoInfo.Ylength;
    Xmax  = Ymax / (1.33333333 * Ymax / ZG_VideoInfo.Xwidth);
    Xinc  = 2.0 * Range / Xmax;
    Yinc  = 2.0 * Range / Ymax;
    Range = -Range;
    for (X = 0; X < Xmax; ++X)
        {
        for (Y = 0; Y < Ymax; ++Y)
            {
            Z = Complex((Range+Xinc * X+Xorig),(Range+Yinc * Y+ Yorig));
        for (I = 0; I < Iterations; ++I)
            {
            switch (Species)
                {
                case 0 :
                    Z = pow(Z,Power) + C;
                    break;
                case 1:
                    Z = sin(Z) + exp(Z) + C;
                    break;
                case 2:
                    Z = pow(Z,Z) + pow(Z,Power) + C;
                    break;
                case 3:
                    Z = sin(Z) + pow(Z,Power) + C;
                    break;
                }
            if ((abs(real(Z)) >= 10.0) || (abs(imag(Z)) >= 10.0)
            || (norm(Z) >= 100.0))
                break;
            }
        if ((abs(real(Z)) < 10.0) ||  (abs(imag(Z)) < 10.0))
            ZG_PlotPixel(X,Y,0);
        else
            ZG_PlotPixel(X,Y,ZG_VideoInfo.NoColors - 1);
        if (kbhit()) break;
        }
    if (kbhit()) break;
    }
while (!kbhit()) ;
if (!getch()) getch();
ZG_Done();
return 0;
}
```

End Listings

Listing One *(Text begins on page 74.)*

```
/* FFFF.C
    -- causes GP fault in real mode on 286/386, and in Virtual 86 mode
    -- catch it in real mode
    -- can't catch it in Virtual 86 mode
      -- distinguishes between GP fault and IRQ5
*/

#include <stdio.h>
#include <dos.h>
#include "gpfault.h"

void (interrupt far *old)();
cdecl main();

void fini(char *msg, int exit_code)
{
    puts(msg);
    _dos_setvect(INT_GPFAULT, old);
    exit(exit_code);
}

void far my_exit(void) { fini("Bye!", 1); }

/*
    In this tiny program, it is easy to determine whether an INT 0D
    was caused by GP fault or by IRQ5 -- just see if CS on the
    interrupt handler's stack corresponds to this code segment.
    A real program would need to do a more thorough check to see
    if r.cs:r.ip corresponded to any known places in the code where
    a GP fault is expected or allowed to occur, then the handler
    could pass the interrupt along using _chain_intr(old).
*/
#define IS_GPFAULT(cs, ip)  ((cs) == FP_SEG(main))

void interrupt far handler(REG_PARAMS r)
{
    if (IS_GPFAULT(r.cs, r.ip))
    {
        printf("\nProtection violation at %04X:%04X\n", r.cs, r.ip);
        /*
            Change CS:IP on stack so control is "returned" to my_exit;
            this is an alternative to using longjmp().
        */
        r.cs = FP_SEG(my_exit);
        r.ip = FP_OFF(my_exit);
    }
    else
    {
        /* handle IRQ5 -- unfortunately, _chain_intr() not in Turbo C */
        _chain_intr(old);
    }
}

cdecl main()
{
    int *p = (int *) -1;
    old = _dos_getvect(INT_GPFAULT);
#ifndef CRASH_AT
    _dos_setvect(INT_GPFAULT, handler);
#endif
    printf("int at %p is ", p);
    printf("%04X\n", *p);
    /*NOTREACHED on 286/386 */
    fini("Done!", 0);
}
```

End Listing One

Listing Two

```
/* GPFAULT.H
    -- REG_PARAM structure represents stack at entry to interrupt handler
    -- CPU pushes flags, CS:IP, and, for protected-mode INT 08-0D,
       an error code
    -- Compiler pushes all other registers at entry to interrupt function
    -- Turbo C pushes registers in a strange order
    -- Watcom C 386 7.0 also pushes FS and GS registers (unfortunately
       MetaWare High C for MS-DOS 386 1.5 does not)
    -- replace Microsoft C 5.1 FP_SEG, FP_OFF macros with ones that don't
       requires lvalues
    -- keep Watcom C 386 7.0 FP_SEG, etc. -- these work for 48-bit pointers
    -- for MetaWare High C for 386 MS-DOS, need our own 48-bit FP_SEG, etc.
*/

#ifdef InstantC_16M
/* Rational Systems Instant-C/16M protected-mode C interpreter */
#define DOS16M
#define PROT_MODE
#endif

typedef struct {
#if defined(__WATCOMC__) && defined(__386__)
    unsigned gs,fs;
#endif
#ifdef __TURBOC__
    unsigned bp,di,si,ds,es,dx,cx,bx,ax;
#else
    unsigned es,ds,di,si,bp,sp,bx,dx,cx,ax;     /* same as PUSHA */
#endif
#ifdef PROT_MODE
    unsigned err_code;                          /* for pmode INT 08-0D */
#endif
    unsigned ip,cs,flags;
    } REG_PARAMS;

#ifdef __TURBOC__
#define _dos_setvect(x,y)   setvect(x,y)
#define _dos_getvect(x)     getvect(x)
#endif
#define INT_GPFAULT         0x0D
#define INT_IRQ5            0x0D

/* 386 protected-mode: far pointer is 48 bits; near pointer is 32 bits */
/* thus, 386 pmode near pointer can hold a real-mode far pointer */
#ifdef __HIGHC__
#define real_far            _near
#define prot_far            _far
#define far                 _far
#else
#if defined(__WATCOMC__) && defined(__386__)
#define real_far            near
#define prot_far            far
#endif
#endif

typedef unsigned long UL;
typedef void far *FP;
typedef enum { FALSE, TRUE } BOOL;

#ifdef __HIGHC__
/* use overlay struct: no High C support for 48-bit immediate values */
/* remember that unsigned is 32 bits, short is 16 bits */
typedef struct { unsigned off; short seg; } overlay;

#define FP_SEG(fp)          ((overlay prot_far *) &(fp))->seg
#define FP_OFF(fp)          ((overlay prot_far *) &(fp))->off
#else
#if (!(defined(__WATCOMC__) && defined(__386__)))
/* Microsoft C FP_SEG() and FP_OFF() require an lvalue:  yuk! */
#ifdef FP_SEG
#undef FP_SEG
#undef MK_FP
#undef FP_OFF
#endif
#define FP_SEG(fp)          (((UL)((FP) fp)) >> 16)
#define MK_FP(seg,off)      ((FP)(UL)(((UL)(seg) << 16) | (off)))
#define FP_OFF(fp)          ((unsigned)((FP) fp))
#endif
#endif

#ifdef __HIGHC__
#pragma Calling_convention(C_interrupt | _FAR_CALL);
typedef void (*IPROC)();
#pragma Calling_convention();
#else
typedef void (interrupt far *IPROC)();
#endif
```

End Listing Two

Listing Three

```
/* GPFAULT.C -- for Rational Systems DOS/16M or Eclipse Computer Solutions
(formerly AI Architects) OS/286

for DOS16M:
    if not exist dos16lib.obj cl -AL -Ox -Gs2 -c source\dos16lib.c
    cl -AL -Ox -Gs2 -c -DPROT_MODE -DDOS16M -Zi gpfault.c
    link /co preload crt0_16m pml gpfault dos16lib /noe,gpfault;
    makepm gpfault
    splice gpfault gpfault

for OS/286:
    if not exist \os286\llibce.lib call patch msc51e
    cl -AL -Ox -Gs2 -c -DPROT_MODE gpfault.c
    link gpfault,gpfault,gpfault/map,\os286\llibce;
    \os286\express gpfault
*/

#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>
#include <string.h>
#include <dos.h>
#ifdef DOS16M
#include "dos16.h"
#endif
#include "gpfault.h"

#define IN_MY_CODE          11593
#define IN_USER_CODE        16843
#define IN_HANDLER          40311

unsigned whereami = IN_MY_CODE;     /* need our own protection for this */
jmp_buf toplevel;
jmp_buf toplevel_copy = {0};        /* initialized, in a different segment*/
unsigned legal = 0;                 /* just a legal address to bang on */
void (interrupt far *old_int13handler)();

void interrupt far int13handler(REG_PARAMS r);  /* GP fault handler */
void goto_toplevel(void);       /* longjump out of handler */
void revert(void);              /* restore default handler */
void fail(char *msg, FP fp);    /* fail by calling default handler */

main(int argc, char *argv[])
{
    char buf[255];
    unsigned far *fp;
    unsigned data;

    old_int13handler = _dos_getvect(INT_GPFAULT);
    _dos_setvect(INT_GPFAULT, int13handler);

    printf("'Q' to quit, '!' to reinstall default GP Fault handler\n");
    printf("%Fp is a legal address to poke\n", &legal);
    /* next line helps illustrate limitations of protection */
    printf("%Fp is not a legal address to poke\n", &legal-1);

    setjmp(toplevel);
    whereami = IN_MY_CODE;
```

(continued on page 122)

Listing Three *(Listing continued, text begins on page 74.)*

```
    memcpy(toplevel_copy, toplevel, sizeof(jmp_buf));

    for (;;)
    {
        printf("$ ");
        *buf = '\0';
        gets(buf);

        if (toupper(*buf) == 'Q')
            break;
        else if (*buf == '!')
        {
            revert();
            continue;
        }

        sscanf(buf, "%Fp %u", &fp, &data);
        whereami = IN_USER_CODE;
        *fp = data;     /* the crucial line of code */
        printf("poked %Fp with %u\n", fp, *fp);
        whereami = IN_MY_CODE;
    }

    revert();
    puts("Bye");
    return 0;
}

void revert(void)
{
    _dos_setvect(INT_GPFAULT, old_int13handler);
}

void fail(char *msg, FP fp)
{
    (fp) ? printf(msg, fp) : puts(msg);
    revert();
    _chain_intr(old_int13handler);
}

void goto_toplevel(void)
{
    if (memcmp(toplevel, toplevel_copy, sizeof(jmp_buf)) == 0)
        longjmp(toplevel, -1);
    else
        fail("Toplevel context has been trompled", 0);
}

void interrupt far int13handler(REG_PARAMS r)
{
    switch (whereami)
    {
        case IN_HANDLER:
            fail("\nDouble fault at %Fp\n", MK_FP(r.cs, r.ip));
            /*NOTREACHED*/
        case IN_MY_CODE:
            /*
                In a program that had to look out for IRQ5 in
                addition to GP fault, here one might want to
                _chain_intr(old_int13handler) and then recover.
            */
            fail("\nInternal error at %Fp\n", MK_FP(r.cs, r.ip));
            /*NOTREACHED*/
        case IN_USER_CODE:
            whereami = IN_HANDLER;
            _enable();  /* reenable interrupts */
            /* we could use Intel LAR and LSL instructions here to
            figure out why GP fault took place:  did we try to write
            into code?  or did offset overrun segment limit? */
            printf("\nProtection violation at %04X:%04X\n", r.cs, r.ip);
            if (r.err_code)
                printf("Error code %04X\n", r.err_code);
            printf("<ES %04X> <DS %04X> <DI %04X> <SI %04X>\n",
                r.es, r.ds, r.di, r.si);
            printf("<AX %04X> <BX %04X> <CX %04X> <DX %04X>\n",
                r.ax, r.bx, r.cx, r.dx);
            goto_toplevel();
            /*NOTREACHED*/
        default:
            whereami = IN_HANDLER;
            _enable();
            puts("whereami flag got trompled");
            goto_toplevel();
            /*NOTREACHED*/
    }
}

#ifdef DOS16M
/*
    NOTE!  For capturing most interrupts in DOS/16M, the following is
    _not_ necessary.  In the vast majority of cases, Microsoft's
    _dos_setvect() and _dos_getvect() will "do the right thing" under
    DOS/16M.  But capturing INT 0D to get at protected-mode GP Faults
    (rather than getting at real-mode IRQ5) is an admittedly uncommon
    case, so here we have to kick down and use the DOS/16M API.
*/
void _dos_setvect(unsigned intno, IPROC isr)
{
    D16pmInstall(intno, FP_SEG(isr), FP_OFF(isr), NULL);
}

IPROC _dos_getvect(unsigned intno)
{
    IPROC isr;
    D16pmGetVector(intno, (INTVECT *) &isr);
    return isr;
}
#endif
```

End Listings

Location is Everything!

A locator for embedded systems programming

Mark R. Nelson

A few years ago, most embedded controllers were 4- or 8-bit microcontrollers such as the 8048, 8051, 68HC11, or COPS-400. But the popularity of the PC has brought the 8088 into this arena for two reasons. First, the price of 8088 processors and support chips was driven down to levels where they can compete with microcontrollers. Second, just as important, sophisticated 8088 software development tools are widely available at low prices.

For example, a typical price for an 8051 compiler/linker/locator package can easily run over $1000. This compares with street prices of under $100 for PC-based C compilers. The support tools (CodeView or Turbo Debugger, for instance) for the PC-based compilers generally outperform those for the microcontroller cross compilers. Even more important, there exists a huge group of programmers that are familiar with the PC-based development tools.

But just because you have a copy of Turbo C or Microsoft C doesn't mean you are ready to go out and start developing code to run an atomic toaster. There is still one key ingredient missing in your software development environment. The missing link is a piece of software called a "Locator," which takes the output from the linker and moves code and data around so that they match up with the RAM and ROM in the target hardware.

A simple 8088-based system will generally have EPROM in high memory and RAM in low memory. This mode of operation is more or less dictated by the architecture of the processor. You have to have ROM or EPROM in high memory because that is where the 8088 begins executing code when it powers up. Because the interrupt vectors for the part are in low memory, hardware designers typically place the system RAM there. So, in a system with 64K of RAM and 64K of EPROM, the RAM will have a base address of 0000:0000, while the EPROM will have a base address of F000:0000.

EXE File Structure

Though it may not be obvious, MS-DOS does, in fact, provide a locator. When COMMAND.COM loads an EXE file, it needs to locate the file at the currently available location in the RAM of the PC. This location changes up and down depending on many factors (the version of DOS, whether or not TSRs are loaded in, and so on). Thus, the EXE file has all the information in it necessary to inform DOS about what it needs to do to run the program anywhere in memory. This information is stored in what is known as the "header" of the file. The structure of the EXE file is shown in Figure 1, and a detail of the header structure is shown in Figure 2.

As Figure 1 shows, the code image is just one of four sections in the EXE file. The code found in that section is ready to be loaded into RAM and executed with no changes, provided it is loaded starting at location 0000:0000.

In order to load it into a different section, the loader needs to go into the code image and adjust any segment references upward by the value of the load segment.

A general-purpose locator should also be able to place code anywhere in memory. But to complicate matters, it needs to be able to split up the code and data sections of the program, locating them in arbitrarily different sections of memory. It needs to be able to adjust any segment references in the code image. The locator also needs to be able to produce a located output file suitable for loading into an embedded system and it needs to provide some

Mark is a programmer for Greenleaf Software, Inc., Dallas, Texas. He can be reached through the DDJ *office.*

sort of initialization so that our software in the target will begin execution on when the processor is reset.

A Locate Program

The program LOCATE (see Listing One, page 152) accomplishes the just mentioned goals. All that needs to be done is to include the startup module, START.ASM (Listing Two, page 153), as the first file in the link. The file can then be run through LOCATE. This produces a HEX output file suitable for loading into an embedded system.

The first thing LOCATE does in order to produce the HEX output file is to read in the header of the EXE file. This is done in the module *read_header_data( )*. The header is read into the *exe_header* structure defined at the top of the file. Once the header is read in, most of the information needed to locate the program is ready. *read_header_data( )* also makes a few more calculations before exiting. First, it computes the offset to the start of the actual code in the EXE file. The header contains the offset in paragraphs, which just has to be multiplied by 16 to create an offset in bytes.

Next, LOCATE calculates the size of the program. This is somewhat more complicated, involving three different numbers in the header. Finally, it determines where in the program the STACK segment is located. This point will mark the line of demarcation between RAM and ROM. As you will see later, the routine START.ASM orders the segments in such a way that this is the case.

After LOCATE has determined where the code image is, and how long it is, it calls the routine *read_input( )*, allocates a buffer, then loads the code image into the buffer. This code image is ready to run as is on an 8088 if it is loaded in at address *0000:0000*.

After reading in the image, all that is left to do is to process all of the relocation table entries. The header has two items that say where the relocation table is in the EXE file, and how many entries are in it. Each entry in the table is just a segment:offset pair, specifying a long address in the code image that needs to be relocated.

The procedure called *process_relocation_table( )* is a loop that gets each relocation table entry, computes where that is in the code image, then modifies the value so that it conforms with the target hardware. The item pointed to by the relocation table entry is a segment value in the code image. For example, when a program starts, there is often a statement that looks like this:

```
MOV    AX, XXXX
MOV    DS, AX
```

The relocation table points to the *XXXX* value, so that it can be modified to reflect the real memory address that should be loaded in. The MS-DOS loader has a simple job here. It just adds the base segment address of the program to the segment in memory. For example, if the code is being loaded in at 2134:0000, it adds 2134 to *XXXX* and stores the new value in the code image.

A general-purpose locator should also be able to place code anywhere in memory

Our locator has to make a slightly more complicated decision. It needs to look at the value of the segment to be relocated. If the segment value is less than that of the stack segment, it means that it is a ROM value, and should be recalculated to go into high memory. If the segment value is greater than, or equal to, that of the stack segment, it should be recalculated to go into low memory. Remember, that the stack segment value is in the EXE header, and we have defined our segments so that all code is below the stack segment and all data is above it.

The relocation values in LOCATE are hardcoded as *F000H* for the program, and *40H* for the data. The 8088 has 256 interrupt vectors that go from address *0* to *3FFH*, so that the first available segment is at *40H*. After the relocation process is complete, the image is ready to be output to the HEX file.

The most common format for transferring code or data to an embedded system is to use Intel MCS-86 Hex code, and that is what this program does. Virtually every emulator and debugger will support downloads in this format. Because it is the closest thing to a universal format, it is used here. Before the code image is sent, a single record is sent specifying which segment the load is to go into. In this case, that will be *F000H*. The image is then formatted into records, and output — one by one — until they have all been issued.

Finally, LOCATE outputs a single JMP instruction that will be located at *FFFF:0000*. This is the restart location for the 8088 processor. When the processor powers up, or receives an external reset signal, control is transferred to this location and execution begins. Because both the code segment and initial IP values are found in the EXE header, they can be output as part of the jump instruction, causing it to automatically point to our startup code.

The last detail at this point is to output a single EOF record in Intel hex format. The code is at this point ready to be sent to our target hardware.

The Startup Code

Although LOCATE will try to relocate any EXE program in any language, it will not produce useful code without a few items being taken into account. Most of them are covered by the inclusion of a startup file, START.ASM. When compiled, START.OBJ will replace the startup code produced by the C compiler, usually found in CRT0.ASM.

The first problem is the ordering of the segments. As mentioned previously, the only reference point available in

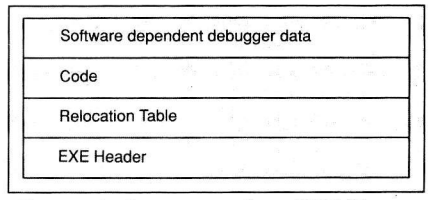


Figure 1: *Structure of an EXE file*

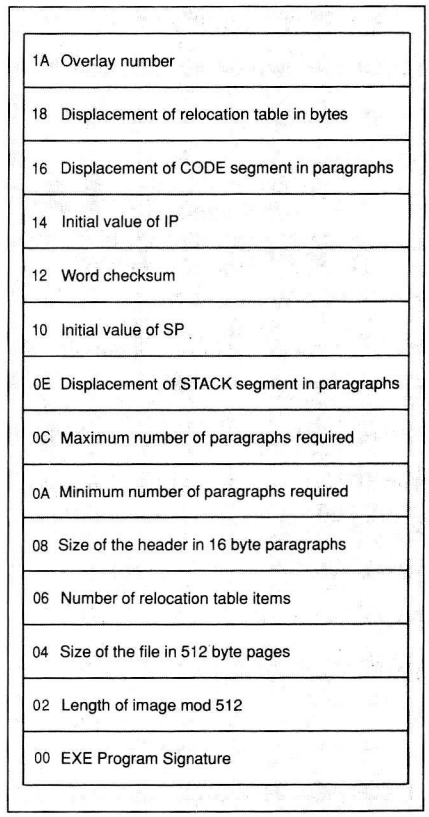


Figure 2: *Structure of the header*

the EXE file regarding the location of data in the EXE file is the stack segment value. This means that we need to force the stack segment to be the first segment in RAM. The sample file shown for START.ASM does this by listing the segments used in a C program in the desired order. By having START.OBJ be the first file read in by the linker, the segment order shown will be the one used. In START.ASM, the DGROUP segment is defined to contain the stack, so the stack segment listed in the EXE header will be the first segment after the code segments. The example shown in the next section of this article will work for Microsoft C or Turbo C. Turbo C does not use a *CONST* class of segment, but will not object to the presence of one.

A "Locator" takes the output from the linker and moves code and data around so that they match up with the RAM and ROM in the target hardware

A second problem for embedded systems is how to handle initialized data. In just about any program there will be some data in the *CONST*, *BSS*, and *DATA* segment types. Loading them into RAM would work, but only once. If the system were to lose power, the code would still be present in the ROM when restarted, but all of the downloaded data would have vanished.

This version of LOCATE and START .ASM handles this in a straightforward way. All of the initialized data in the image can be found immediately after the code segments. The initialized data values in the image are loaded into ROM just as they are found in the code image. When the program restarts, control is transferred to START.ASM. All it does is make a copy of all the initialized data from ROM into RAM. This actually takes less space in the ROM than would be used to initialize them using code.

A few precautions are needed to create code that will actually work under this system. First, some compilers will generate code that performs stack checking each time a function call is invoked. Because we are tinkering with the stack segment, this code frequently will not work. Both Turbo C and Microsoft C allow you to disable stack checking with command line switches, and these should be used.

C library routines should always be viewed with distrust. Some of the library routines perform DOS calls, and these will obviously not work on an embedded system. Others have built in linker commands that will reorder your segment definitions. Unfortunately, the only way to tell if a library routine will work is usually to just try linking it in. Microsoft C will generate in line code for many library routines, and these will almost always work. Once again, this can be invoked with command line options.

An Example

A sample program that uses this system is shown in Listing Three, page 153. The program performs the standard first function of a C program, which is to print simply a "Hello, world!" message on the screen. Naturally, *printf* will not work in my embedded system, so it has been replaced with *my_printf( )*.

The function *my_printf( )* is writing characters to an imaginary printer port that has to be polled before it can be written to. It calls the library routines *outportb* and *inportb* to access the hardware. The only predefined data it will have will be the string "Hello, world!\n". Using Turbo C, this program can be compiled with the command line:

```
TCc -ms -c HELLO.C
```

The Microsoft C command line would be:

```
CL /AS /C /Oi /Gs HELLO.C
```

The Microsoft C command line specifies no stack checking, as well as generation of intrinsic functions.

After compiling, the code would be linked with the line:

```
TLINK START.OBJ+HELLO
.OBJ,HELLO.EXE,HELLO.MAP,  \TC\
                          LIB\CS;
```

or

```
LINK START.OBJ+HELLO.OBJ,
     HELLO.EXE,HELLO.MAP/MAP;
```

The resulting map file should look similar to the one in Figure 3. This shows that the program and startup code will be located first, followed by constant data, the uninitialized data, and the

(continued from page 126)
stack. The program is then located using the following command line:

```
LOCATE HELLO.EXE HELLO.HEX
```

This produces an Intel hex file that looks like the one in Figure 4. This program is ready to be loaded and executed in the target system. The only real complication to debugging at this point is the relocation of symbol values. The offset values in the map will all be correct, but the segments will all be wrong.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 152.)

Vote for your favorite feature/article.
Circle Reader Service **No. 7.**

Start	Stop	Length	Name	Class
00000H	0008DH	0008EH	_TEXT	CODE
00090H	000B4H	00025H	END_OF_ROM	STARTUP_CODE
000C0H	000C0H	00000H	_CONST	CONST
000C0H	000C0H	00000H	_BSS	BSS
000C0H	000CFH	00010H	_DATA	DATA
000D0H	002CFH	00200H	_STACK	STACK

Program entry point at 0009:0000

Figure 3: *Map file after compilation*

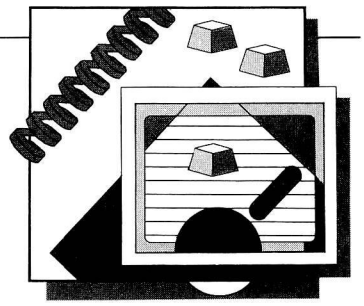
```
:02000002F0000C
:22000000B8060050E8060059EB00EBFEC3558BEC568B7604EB40803C0A750FEB00B8000250E879
:22002200450059A9800074F3B80D0050B8000250E84D005959EB00B8000250E8290059A9800000
:2200440074F38BDE468A079850B8000250E82E005959803C0075BB5E5DC3558BEC8B5604EDEB46
:22006600005DC3558BEC8B5604EC32E4EB005DC3558BEC8B56048B4606EF5DC3558BEC8B560452
:220088008A4606EE5DC30000B80BF08ED0BC00088CC80503008ED8B80BF08EC033F6B9FFFBF30B
:1B00AA00A48CC08ED8FBEA000000F00048656C6C6F2C20776F726C64210A007D
:02000002FFFFFE
:05000000EA000009F018
:00000001FF
```

Figure 4: *Contents of hex file*

ARCHIVES

Realizable Fantasies

DDJ will continue the active pursuit of 'realizable fantasies' that are within the bounds of current technology and knowledge. This includes computer music, real-time video graphics, and unusual input techniques. We will also explore shared memory, computer networking via telephone, and who knows what else. You are part of this. The *Journal* is primarily a communication medium and an intellectual rabble-rouser. Send us your ideas, your creations, your problems, and your solutions. The more we all share, the more we all gain. — DDJ *Editorial, March 1976.*



A Taste of Lisp

Michael Swaine

Over the next months I'll be looking at Lisp, Common Lisp, and recent developments in Lisp programming that make the language an interesting option for professional programmers. This month, I'll prepare the ground by taking a look at Lisp in general, debunking some myths, giving a sense of what it's like to program in Lisp, and sketching the history of the language, leading to the establishment of the Common Lisp standard.

Time for Lisp

One of the programming paradigms I wanted to explore when I conceived this column was interactive recursive function-based list processing: Lisp. Learning Lisp back in the 1970s was a revelation for me, opening my eyes to the arbitrariness of the Fortran/Basic/Pascal approach I had naively thought was the one true methodology of programming. Lisp was really a different paradigm, requiring a different way of thinking about problems, and embodying different ideas about the value of memory and time.

That reference to the value of memory and time was intended to be only partially ironic. Lisp implementations of the 1970s generally were written by people who believed that memory was cheap and was getting cheaper, and who valued development flexibility over execution speed — these were implementations that ate cycles and RAM. But there is time and there is time. While the Lisp systems of the 1970s were not generally fast number crunchers, they were demonstrably good for developing highly-complex applications, and some Lisp-based numeric applications of this era probably could not have been developed in, say, Fortran in any acceptable span of time. Lisp became the language of choice for artificial intelligence work, but for years, as a result of execution-time slowness, memory hungriness, and a proliferation of incompatible dialects, it could not be considered seriously for commercial applications. Recent developments, though, have made Lisp a language worth any serious programmer's attention.

One of these is the increasing power of personal computers. With brief exceptions, memory has indeed continued getting cheaper, and systems have been getting faster and more powerful. Another development is the work that has gone into creating efficient Lisp implementations. Some of this work has been tied to specific hardware, but progress has also been made in Lisp implementations on standard hardware. Together, these developments have been whittling away at the stumbling blocks between Lisp and the non-academic programmer.

But arguably the most important event in Lisp's history is the definition and acceptance of the Common Lisp standard, which has probably saved the life of the language.

This is a good thing, because there is a whole list of problems for which Lisp is the best language in existence. I'm pretty sure there is, anyway, although I couldn't prove it; probably nobody could. Wade Hennessey, in his book *Common Lisp* (McGraw-Hill, 1989), runs down the somewhat longer list of problem areas in which, for some people at least, Lisp is the language of choice. His claim is a provable and correct version of my unprovable assertion. His list includes: Emulating human vision, doing symbolic algebra, developing special-purpose languages, exploring natural language understanding, theorem proving, planning complex actions, computer-aided design, programmer education, and even writing text editors (such as Gnuemacs) and doing system programming (at least on Lisp machines).

Pulling together a standard for Lisp after 20 years of divergent development was a little like gathering the scattered energy of the Big Bang. Guy Steele found a better simile. He prefaces his definitive *Common Lisp: The Language* (Digital Press, 1984) with a James Madison quotation from *The Federalist* about the difficulty of hammering out consensus on the Constitution. It took a kind of electronic Constitutional Convention to unite the rebel colonies of Lisp development.

Lisp had grown, after its creation around 1960 by John McCarthy, in many directions, the most important in the 1970s being the MacLisp, InterLisp, and Scheme dialects. By 1980, Scheme had begotten T, MacLisp had inspired NIL and Franz Lisp, special purpose hardware had given the world Zetalisp and others, and the needs of university researchers had produced SpiceLisp and the wistfully-named Portable Standard Lisp (PSL). Common Lisp was the compromise dialect designed in the early 1980s to be palatable to as many Lisp developers as possible. Common Lisp is not the collection of features that the above-mentioned dialects have in common, as the name might suggest. It is much more a union of features than an intersection.

One of the areas into which Common Lisp extends Lisp is object-oriented programming, particularly with CLOS, the Common Lisp Object System, aka Thmalltalk, about which we'll hear more next month.

What Lisp Is

Some of these characterizations will not seem true of, say, CLOS, but what I'm describing here is the core to which things such as CLOS are extensions.

Lisp is recursive. Is it ever. Data structures and functions are routinely defined through recursion. The professor from whom I learned Lisp told a lot of lies, such as that all of Lisp could be constructed by recursive definitions using just five primitive functions. It's a good lie, because it suggests the degree to which Lisp depends on recursion.

Lisp is function-based. Programming in Lisp is a matter of expressing a problem as a composition of functions. To write a program, you define a function, and the definition consists primarily of function calls, with some scaffolding. (You need a little more than five primitive functions, and not all functions are defined recursively.) Values passed to functions are typically the values returned by other functions, although functions themselves can be passed, just as other data can. The same professor taught that no Lisp function should take more than five lines to write, another useful lie that gives a sense of how the functional style works: "pure" Lisp programming makes less use of structuring tools than Pascal programming, using the function as almost the only structure.

Lisp is LISt Processing. A Lisp program is a list, and the list is the fundamental data structure of Lisp. The obvious implication of these two statements is that Lisp programs are Lisp data, and this is in fact a defining feature of the language. It makes it easy for one program to read, create, or execute other programs. It's the source of the great extensibility of the language, which is in turn the source of the plethora of idiosyncratic versions that sprang up and led to a crisis in compatibility, and then to a kind of Continental Congress of Lisp developers in response to the crisis, and ultimately to Common Lisp.

Lisp is interactive. There have been Lisp compilers for a long time, but the interpreter is central to any implementation — for several reasons. Lisp was originally designed to be a language in which provably correct programs could be written, but provability constraints tend to be loosened in the tricks that have to be played to generate efficient compiled code from a Lisp system. Moreover, the kinds of problems to which Lisp is typically applied require a lot of exploratory programming that also makes the interpreter necessary.

The style of interaction is not command action, however, but function call. Because even the main program is a function, invoking a Lisp program is usually a different process from invoking a Pascal program. To invoke a Lisp program, you call the function with appropriate arguments, and it returns a value. The argument list can be empty, in which case calling the function feels more like invoking a Pascal program. Rather than supply the input to a function as parameters, you can write an "Input" function that conducts a dialog with the user to get the input. The value returned can be an arbitrarily complex object, and the function can produce side effects.

So to perform a computation and then print the results in a table, you could write a computational function *ComputeValues*, an input function *GetValues*, and an output function *PrintTable*. *PrintTable* would produce its output as a side effect. You could then perform this function call:

```
(PrintTable (ComputeValues
                    (GetValues)))
```

What Lisp Isn't

Lisp has a reputation. Some of it isn't deserved.

Lisp is not an AI language. It is, however, the premiere language for artificial intelligence programming. Its virtues for AI programming include, but are not restricted to, the ability to manipulate programs as data, which makes it good for symbolic processing of the sort done in symbolic Algebra programs; and its interactive style, which facilitates exploratory programming and prototyping. These virtues are themselves not restricted to the domain of artificial intelligence.

Lisp isn't inherently slow. It has been a prevailing attitude in the Lisp community that performance is not very important, and that's changed, as I hope to show next month.

Lisp isn't a memory hog. There's nothing inherent in Lisp that makes it a memory hog, although past implementations have often been wasteful of memory. The problems to which Lisp is characteristically applied (see Hennessey's list a few paragraphs back) are a different matter. Some of them do require a lot of memory, no matter what language is used to solve them.

A Taste of Lisp

Here's an example that is basically accurate, although it glosses over some complexities.

You define the function using *defun*, which takes three arguments: A string representing the name of the function to be defined, a list of its arguments, and a list that is a function call. This last list is the body of the function being defined. For example:

```
(defun <name> <argument-list>
                        <body>)
```

Here's a recursive definition of a function that, given a nonnegative integer *n*, returns *2* to the nth power:

```
(power-of-two (n)
   (cond
     ((= n 0) 1)
     (t (* 2 (power-of-two (- n 1))))
```

The name of the function is power-of-two, it takes one argument, *n*, and the body of the definition is

```
(cond
   ((= n 0) 1)
   (t (* 2 (power-of-two (- n 1)))))
```

This expression consists of the function (actually a macro) *cond* applied to two arguments. The arguments are

```
((= n 0) 1), and
(t (* 2 (power-of-two (- n 1))))
```

cond is short for conditional, and it takes any number of two-item lists as arguments, returning the last item of the first list whose first item is true. That is, it works like a case structure, picking out a return value based on tests associated with the possible return values. In this case, the first list is *((= n 0) 1)*, and *cond* understands this to mean *"if (= n 0) is true, then return 1."* The expression *(= n 0)* is a call to the function =, which follows Lisp prefix-and-parentheses notation and is a Boolean function that returns *true* if its arguments are equal, and *false* otherwise. So if *(= n 0)* returns *true*, *cond* returns *1*.

If *(= n 0)* returns *false*, then *cond* examines the next expression, *(t (* 2 (power-of-two (- n 1))))*. The first element of this expression is a system constant, *t*, which always returns *true*. This line typifies the syntax of the "else" clause of a *cond*. Because *t* always returns *true*, a *cond* whose last list starts with *t* is guaranteed to return some value. In this case, if the last list is examined, the value returned by *cond* is the value of *(* 2 (power-of-two (- n 1)))*. This is the multiplication function, *, applied to *2* and a recursive call to power-of-two. In the recursive call, power-of-two gets as its argument *(- n 1)*, or *n-1* in mathematical infix form.

In other words, the function power-of-two returns *1* if called with the argument *0*, and for any larger integer re-

turns *2*power-of-two(n-1)*. The definition ensures that, for proper arguments, the recursion will terminate.

I worked through this example in excruciating detail because it gives a strong taste of "pure" Lisp. Lisp isn't as pure as it once was, and in one sense that makes it easier to learn. Other languages employ recursion, for example, and Lisp employs structures that avoid some of the costs of deeply-recursive function definitions.

But this similarity of Lisp to other languages can also, paradoxically, make it harder to understand Lisp. Common Lisp has features that will be familiar to Pascal and Smalltalk and Forth and C programmers, which can lead to thinking of it as something other than what it is, so that its unique features seem like eccentricities. Looking at "pure" Lisp can help in understanding what the language is really about. This example was intended to be a taste of that.

AND (Now for Something Completely Different)

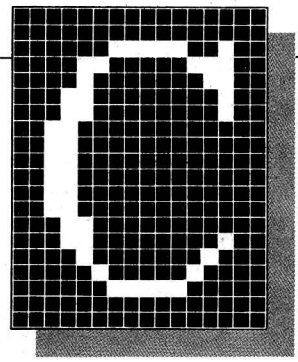
Looking through my "Ancient Programming Languages" file, I come across a description of a language called "AND." AND is a remarkably simple and powerful language, written using just four symbols. The symbols combine in triplets to form twenty commands and three marks of punctuation (there is some redundancy in the coding, obviously). AND is invariably implemented as a one-pass compiler. Despite the language's simplicity, no one yet understands it fully, possibly due to the lack of documentation. Work has begun on the manual, which is expected to run into millions of pages.

AND has been around longer than Lisp or Fortran, and there are a lot of AND programs in use today. Some software systems based on AND have solved difficult calculus problems, piloted rockets, won chess matches, solved crossword puzzles, written novels, graduated from prestigious universities, and managed multinational corporations. Despite these impressive achievements, AND is generally regarded as inappropriate for artificial intelligence programming.

Wait a minute. Did I misspell the name of the language? I'm sure those three letters are in it, but I may have got them in the wrong order.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 9.**



TEXTSRCH Continued: The Expression Interpreter

Al Stevens

Last month we built an expression parser that processes expressions consisting of keywords, Boolean operators, and parentheses. We will use the expressions in text searching queries for a new "C Programming" column project, a text data base indexing and retrieval system. This month we give the name TEXTSRCH to the project, and continue with the expression interpreter.

The text data base that TEXTSRCH will support is one that consists of a large number of text files that do not change much. Typical applications include files of electronic mail messages and engineering documentation. There are many others, of course, but the characteristic they share is that they are relatively static.

The primary purpose for the data base is to store our files of text in a centrally organized place. The primary purpose for TEXTSRCH is to allow us to find the files we want to read, based on some criteria. Given that we know something about what we are looking for, we want to be able to express those criteria in the form of a query expression and to find the files in the data base that match the expression. A typical expression might be:

```
(cobol and fortran) and not pascal
```

A search of the data base would tell us which files match the query, that is, which files contain the words "cobol" and "fortran" but do not contain the word "pascal."

The expression parser from last month did a lexical scan of the expression, converting the expression's words and operators into tokens, and placing them into a stack in postfix notation. This month we will build the TEXTSRCH expression interpreter. Its function is to extract the keywords from the stack, initiate a search of the data base for a match on each one, and combine the results of the search with the Boolean operators to determine which files match the search criteria. After we build such a file list, we will switch to a text scan to complete the search and find the places in the files where the words and phrases appear. Those processes will come in a later installment. The index that we will build allows us to get down to the file level with a minimum of search time.

First let's consider how we will index our text data base. We won't build or use the index in this month's installment, but we must understand its operation in order to build the expression interpreter.

Our data base will consist of a number of text files and a word index. The index will contain an entry for each keyword or phrase that appears in the data base. The index entry for a particular word will tell us which files contain that word by pointing to a file list. The list is recorded as a bit map where each bit represents a file. At any given time, the number of files in the data base is constant, so the length of the bit map for any entry is fixed at that number of bits.

When we search the index for a word, we will get back a fixed-length bit map with those bits turned on that correspond to the files that contain the word. The mechanics of the index and the search are not important to us now; it is necessary for us to know only that a search returns such a bit map.

With such a map we can tell if a word does or does not appear in a file. Suppose our data base has 16 text files in it (not very many, but enough to illustrate the principle). If we use the index to tell us which files include the word "fortran," it might return to us a bit map such as this:

```
0010001011000010
```

Reading the bits from right to left, we can see that the second, seventh, eighth, tenth, and fourteenth files in the list contain the word. In our data base manager, we will have a table of file names that match the bit positions, so by applying the one bits, we can derive the file names where the word appears. Understand that we have not actually read the file to get this information — not during the search, that is. Sometime in the past we read all the files to build the index. This is why such data base architectures work best with relatively static data. The most time-intensive process is the construction of the index. After that, searches can be a snap.

You can see how a Boolean query would work. We will search the index for each word in the query. Then we will apply the Boolean operators to the bit maps that the search returns. The final bit map is the one that lists the files that match the complete search criteria.

You will remember from last month that the query begins life in infix notation that the parser translates into postfix notation. The search itself must be driven by an expression interpreter that reads the postfix stack.

(continued from page 132)

Let's see now how to interpret a postfix expression. Consider a simple expression such as this one

apples and oranges

The postfix stack for this expression would look like this:

<and>
apples
oranges

The interpreter pops entries from the stack and processes each one according to what it is. If the entry is a binary operator token such as the *<and>* shown here, the interpreter pops the next two tokens, processes them, and combines the two operands by applying the operator. If the entry is a word or phrase operand, the interpreter calls the search process to convert the word or phrase into a bit map.

Here is how our "apples and oranges" expression would be interpreted. First the interpreter pops the *<and>* token. Next it pops the *oranges* token and converts it into a bit map. Then it pops the *apples* token and converts it into a bit map. Because the *<and>* token is waiting, the interpreter logically ANDs the two bit maps and returns the result.

To see how the interpreter works with a more complex expression, consider this:

fortran and not (cobol or pascal)

The postfix stack for this expression looks like this:

<and>
<not>
<or>
pascal
cobol
fortran

The interpreter pops the *<and>* token just as it did with the simpler example. This means it expects two more operands to AND together to get the result. The next interim result it must compute is indicated by the *<not>* token that it pops. This token says that the result of the next operand must be the ones complement of the search result. The next token is the *<or>*, which says that two operands must be popped and logically ORed together. The *pascal* and *cobol* operands come next. The interpreter calls the search function for each of these words and gets back two bit maps, which it ORs together. Then the interpreter takes the ones complement of that result, and the result of all that is the first operand required by the *<and>* operator. The next token popped is *fortran*. The interpreter ANDs its bit map with the first bit map, and that delivers the result of the search. Out of breath? What I have just described seems complicated but is a relatively simple algorithm that uses recursion in a function that returns the search result. When it pops an operator, it calls itself to pop the next stack entry or two and returns the result that comes from applying the operator.

Listing One, page 154, is a new version of textsrch.h to replace the one from last month. A notable addition to this file is the MAXFILES global variable. This variable specifies the maximum number of text files in your data base. The *bitmap* structure defines the multiple-integer bit map that represents the search result. We use a structure here so that we can easily pass the map between functions.

Listing Two, page 154, is exinterp.c, the expression interpreter. It expects the postfix expression to have been built by the parser from last month. It returns a bit map that represents the result of the search. We'll discuss more

about the search later. The *exinterp* function calls the recursive *pop* function to get the bit map. The *pop* function is where the real work is done. It pops a token from the postfix stack and processes it. If that token is an operand (a word), the *pop* function calls the *search* function, which returns a bit map. The bit map has a bit set for each file that includes the word. If the token is an operator, the *pop* function calls itself recursively to get the next token or two for the operator to work on. The three functions named *and*, *or*, and *not* modify the bit map to reflect the result of the Boolean operations they represent. We use functions here because the bit map is an array. (Think how much easier this would be with C++.)

The primary purpose for TEXTSRCH is to allow us to find the files we want to read, based on some criteria

Listing Three, page 155, is search.c. This one is a throwaway. Although it works, it is less than efficient. Its purpose is to allow us to test and demonstrate the expression evaluation algorithms. It includes three generic functions that we will replace later. The first one, called *init_database*, initializes the data base system. For now, all it does is build a list of file names that have the .TXT extension. This stub function uses MS-DOS file structures and the Turbo C *findfirst* and *findnext* functions. If you are using a different environment, the substitutions should be simple. What you want is a list of all the file names in the data base built into the array called *names*.

The second generic stub function is called *search*. Its purpose is to search the data base to determine which files have the specified word in them. This one cheats. It calls the *grep* program and redirects its output to a file called "hits." The *grep* command that it executes looks like this:

```
grep -l -i <word> *.txt >hits
```

This format works with the Turbo C *grep* program and should work with most others. The -l parameter tells *grep* to list only the file names that match the search, and the -i parameter tells it to make the search case insensitive. When the *grep* program is done, the *search* function reads the hits file and compares its entries to the file name list that *init_database* built. For those files that match, the *search* function sets a bit in the bit map that it builds and returns.

The third generic stub function in search.c is called *process_result*. It accepts a bit map as the result of an expression search. Remember that *search* builds a bit map for a single word, and *exinterp* combines all the bit maps with the operators to build a final bit map. That final one is the one that *process_result* gets. In this stubbed version, *process_result* simply lists the files that are found to match the search criteria.

Listing Four, page 156, is textsrch.c, the main function of the TEXTSRCH program and the one that ties the search process together. It calls *init_database* and then reads the search expression from the console. It calls *lexical_scan* from last month to convert the expression to postfix notation. If the expression is in error, the program displays a caret under the token where it found the problem. Otherwise, it calls *exin-*

terp to interpret the expression and *process_result* to process the result.

You need to compile express.c from last month and textsrch.c, search.c, and exinterp.c from this month to build the program. Make sure that you use the new textsrch.h from this month.

To run the program, build your data base first. Copy all of your text files into a single subdirectory and give them the file extension .TXT. If the files have word processor stuff embedded in them, run them through your word processor to build straight ASCII text files. Otherwise, *grep* might not be able to find the words it searches for.

These instructions assume an MS-DOS environment. Make sure that the *grep* and *textsrch* programs are in the path and that you are logged onto the subdirectory where you built the data base. Run the *textsrch* program from the command line. It will prompt you for an expression. Type one in and watch it churn. If your data base is big, it will take a while. Later, after we build an index, the search will be much faster. The result of the program will be a displayed list of files that match the query. You can verify the results with *grep*.

Next month we'll work on building an index into the data base. Things get interesting because we have several different ways to approach the problem.

OOPSLA

I attended the ACM's fourth annual Convention On Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), held in New Orleans in October, 1989. This was my first trip to the Crescent City, a rare confession from an old jazz musician, and besides the hoopla of OOPSLA, I heard plenty of good jazz and ran into some old friends, both on the conference floor and on bandstands throughout the town.

C++ permeated the show. This is clearly the platform of the very near future. There were demonstrations of C++ language environments from ParcPlace and Apple as well as a number of products that enhance the C++ platform. Absent were Zortech, Intek, and Guidelines, the main purveyors today of complete C++ systems for the PC. We can expect announcements soon, I hope, from some of the other vendors.

One subject popped up frequently in conversations with the C++ language vendors. There is no standard class library, and no one seems to know where the first one will come from. The attitude seems to be that users of the language will each build their own. This reminds me of the early days of C

when the few functions described in K&R constituted the only standard we had, and the compiler vendors extended it with their own unique versions of the things left out. There was the Unix library to go from, but many of the PC compilers went their own ways. Market pressure enforced compliance with a de facto standard long before the ANSI X3J11 committee was close to a finished draft. That pressure was created by the overwhelming success of the Microsoft C compiler. Every other compiler vendor tried then to be compatible with good old MSC, and a PC standard of sorts evolved. Some of that standard — the pieces that were not PC specific — found its way into the ANSI draft.

Right now there are several books about C++. Most of them describe some specific classes that they use to explain how you build classes rather than to advance a standard for a class library. I did the same thing myself a few columns back with homegrown classes for windows, menus, strings, and linked lists. None of these disparate libraries are likely to drive an industry-accepted standard. I suspect that the first compiler vendor to bundle a class library with a highly successful C++ compiler will hold the lead.

Bjarne Stroustrup makes the point that we mustn't think about a universal class library the way the pure object-oriented data model would have us do. Classes defy universal definition because the universe is too big. Remember that the purpose of the standard C library is to define methods for manipulating generic objects: Files, strings, characters, numbers, memory blocks, program execution, dates, times, and so on. A standard class library must restrict itself as well to classes that have general application, and the hierarchy for a given program is not necessarily a subset of a universally defined hierarchy but rather one contrived to suit the problem at hand. Eventually the builders of vertical applications might come to share their work in a series of application-specific class libraries. I doubt it, though. Few such industry-wide sharings of C function libraries exist.

The Teaching of C

Another frequently-expressed sentiment at OOPSLA was that C++ is too hard to learn, and the language industry fears that programmers will be scared off by this. They compare C++ to one of the new Pascals with object-oriented extensions and conclude that the Pascal route is easier to travel. They are right to some extent. C++ is more difficult to learn than object-oriented Pascal. But then again, traditional C is more difficult to learn than traditional Pascal, yet C became and remains the dominant language. Why? Because, once learned, C becomes the programmer's preferred language, religious language wars notwithstanding. I think that C is more difficult to learn only because it is more difficult to teach. We need to educate the teachers first. They need to know how to present C (and C++) to the student in logical increments.

There is no standard C++ class library, and no one seems to know where the first one will come from. The attitude seems to be that users of the language will each build their own

C teachers should be required to teach Pascal first. Maybe its syntax will help them understand the properties of a programming language whose original purpose was for teaching programming. C instructors around the country are bewildering students with abstruse code examples the likes of which only a terse language such as C will permit. It isn't that you have to write C programs that way, it's just that you can, and the teachers haven't learned to leave that part until later.

One of the first lessons in K&R is about how you can say this:

```
while ((c = getchar( )) != EOF)
. . .
```

Many teachers conclude that this facility of C to support embedded expressions is so powerful that it needs to be taught first thing out of the chute. There is no compelling reason to avoid such expressions once you understand them, but there is also no reason why a newcomer to C shouldn't be spared these details until he or she is ready for it.

The K&R book precedes this example with a less terse one that breaks each of the expressions and operations

into individual statements like this:

```
c = getchar( );
while (c != EOF) {
    . . .
    c = getchar( );
}
```

The purpose of this earlier example is to introduce the experienced programmer to the terser style of code and to illustrate how every statement is an expression and that you can use that feature of C to write fewer lines of code.

Upon seeing such a program, a C veteran is immediately moved to implode everything until it looks like the first example. But the rookie finds the more verbose program a lot easier to comprehend. The student sees a working program before learning a lot of the finer points of C. The extent to which a teacher should delay some of the finer details will depend on the programming experience of the new C student, particularly because more and more curricula are catering to students who pass progressively through layers of prerequisite classes without logging any real-world programming time. Try, for example, to explain to someone who is not a programmer why recursion is important.

Another problem is that teachers teach too many other complex topics in what are supposed to be C courses. They strive to shroud C in mystery. I recently tutored a student who was enrolled in an advanced C course at a community college. The programming assignments required her to use the low-level I/O and interrupt functions of Turbo C to manipulate the sectors of a DOS diskette directory. The class spent more time learning about the internals of DOS's file system than about C. The instructor's rationale for such assignments was that C is used primarily for systems programming, and DOS is the system that the school uses. A student who is going into a non-DOS C environment has been cheated by this approach. To begin with, the instructor's assumption is no longer valid. We use C to develop nearly every kind of software system. By spending so much of the class's time and energy on the learning of DOS and BIOS sector I/O, the instructor has lost valuable opportunities to teach C. This class would have been more appropriately labeled an MS-DOS Systems Programming Course with C as the programming language. Fewer students would have signed up, of course, but the ones who did would have gotten what they paid for, and the others could have been elsewhere in a real C class.

The ANSI Corner

Sometimes an improvement can get in the way. The draft ANSI specification for standard C adds adjacent string literal concatenation to the language, and that is a handy thing to have. With this feature you can code adjacent literals, and the compiler will concatenate them. For example, consider this code:

```
printf( "Hello," "World");
```

When the compiler sees adjacent string literals such as these, it combines them. The power of this facility becomes apparent when used along with the preprocessor in this way:

```
#define PROGRAM "Jiffy SpreadSheet"
#define VERSION "v1.23"
#define DATE "1990"

printf(PROGRAM VERSION "
                    Copyright " DATE);
```

This feature, handy as it is, cost me a bit of time. I had the following array in a program:

```
char *names[] = {
    "add",
    "change",
    "delete",
    "cut"
    "paste",
    NULL
};
```

See that missing comma after the "cut" literal? I didn't, not for a long time, and I couldn't figure out why this statement worked only some of the time:

```
if (strcmp(names[i], name) = = 0)
```

I could add, change, and delete, but I couldn't cut or paste. After I found the problem, I stared in disbelief, wondering why the compiler hadn't warned me about the missing comma. Then it hit me. Good old ANSI C thought I really meant to build a "cutpaste" string. But, when viewed in the context of how I coded the array, the missing comma was hard to spot, and, once revealed to the old eyeball, looked like it should have been flagged by the compiler as an error. Even after I saw the error, it was not immediately obvious to me that I had coded perfectly acceptable ANSI C language. The compiler, however, looks dispassionately at our code and cannot guess our intentions. What I coded was legitimate C, yet it didn't work worth a hoot.

Here is another use for the # preprocessor operator that we discussed last month. Suppose you have some maximum number of things your program will handle and you define that maximum this way:

```
#define MAXTHINGS 500
```

When the user tries to add the 501st thing, you want to display a message that specifies the limit. We used to do it this way.

```
printf("%d entries only", MAX);
```

This is a contrivance of constants. MAXTHINGS is a constant, and should be included in the string literal. But pre-ANSI C had no convenient way to do that and still preserve the global definition of MAXTHINGS. With the new # operator and adjacent string concatenation, we can do this:

```
#define str(x) #x
printfSTR(MAX " entries only");
```

Book of the Month

Get yourself a copy of *C Traps and Pitfalls* by Andrew Koenig (Addison-Wesley, 1989). It is a collection of the mistakes a C programmer makes. Besides the obvious ones (the dangling *else*, for example), the book contains many insightful discussions of less obvious but common errors including programmer-induced malfunctions of the dreaded C pointer.

Many of the errors that Koenig describes will be flagged as warnings by contemporary ANSI conforming compilers. But a lot of pre-ANSI code is still out there in maintenance, and we must either turn the warnings off or disregard them, and so we are sometimes susceptible to these errors now just as in the old days.

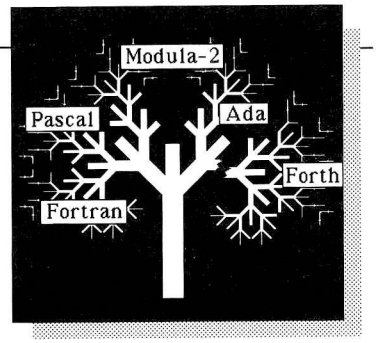
Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 154.)

Vote for your favorite feature/article.
Circle Reader Service **No. 10.**



Dressing the Truth In Rubber Suits

Jeff Duntemann, KI6RA

Remember AI? Sure you do. A couple of years ago, it was going to change the face of the planet, according to the legions of ignorant Esther Dyson wannabees selling $900 newsletters to corporate seatwarmers with lots of money to waste. AI was going to allow programs to configure themselves. AI was going to allow us to get our work done without expending any effort. AI was going to allow our software to read our minds. It got so bad that *PC Week* columnist Jim Seymour suggested that software vendors could double sales by slapping a gold starburst sticker on the outside of every product package reading, "New! Improved! With AI!"

Meanwhile, back in the trenches, the guys who were researching AI were sadly shaking their heads. They never made those promises, or any promises. They weren't trying to create an artificial human brain. They were just looking for new ways to arrange the same old instructions we've been fetching and executing all along. The human brain is a pretty successful computer, they reasoned, so why not try to learn something from the way it works? Had they called it cognitive modeling or somesuch, nothing dramatic would have happened. (Industry mavens rarely abuse what they can't pronounce.) But say "artificial intelligence" and, whammo! Isaac Asimov's robots come striding over the horizon, with *Popular Mechanics*, *People Magazine*, and finally *The National Enquirer* in hot pursuit.

You'll notice that nobody's talking much about AI anymore. The hypemongers buried it so deep that nobody in the mainstream software development community may ever take it seriously again. The hype machines have been quiet lately, but don't assume that they're gone — they're just getting a valve-and-ring job so to be in top shape when the next fad happens by, like remoras watching for a passing grouper.

Lately, I've begun to hear them revving, this time over something with a lot more near-term potential than AI: Object-oriented programming. People who never said OOPs before, except when using chopsticks, have become self-anointed experts, and have begun dressing the truth in rubber suits.

So let's dump some sand in the hype machines, and nail some nascent conventional wisdom to the wall.

Rubber Truth #1: OOP Makes Coding Effortless

Dream on. Furthermore, OOP doesn't even make coding easy. One might argue that it makes coding more difficult, in that OOP requires considerable forethought and design effort up front.

An object hierarchy requires a starting point; a handful of abstract classes that exist to broadcast certain characteristic behavior down the many inheritance branches. The root of the tree is a foundation on which thousands or tens of thousands of lines of code may depend. If you discover halfway through a 50,000 line application that you've conceptualized the fundamental abstract classes wrong, you might just have to kiss 25,000 lines of code good-bye. So take heed:

Duntemann's OOP Warning #1: Invalidating the Root Invalidates The Leaves!

When you cast a program design in OOP terms, think very hard about the nature of your conceptual model. Once you've created a solid, correct, and workable foundation in terms of an object hierarchy, coding gets a little easier because you can inherit and reuse some of that hard, early general work in the later, more specific tasks. That doesn't mean that the total effort required to finish the project is going to be any less. It's just that the hardest work gets done up front, and the rest of it seems easy only by comparison.

Rubber Truth #2: OOP Makes Programs Bulletproof

I confess, I've stretched this one a little myself, and have been bitten in the hindparts for my trouble. At first thought, encapsulation might appear to make it harder for bugs to propagate beyond the bounds of the object in which they occur. Unfortunately, all method code is bequeathed to an object's children, for good or for bad. There's no software equivalent of Maxwell's Daemon sitting on a post at the object's interface, letting features pass down the object hierarchy but keeping the bugs behind. So, folks, keep in mind:

Duntemann's OOP Warning #2: Bugs Are Just as Inheritable as Features

It's true to an extent that languages that enforce encapsulation, such as Smalltalk and Actor, prevent a certain class of under-the-table bugs caused by directly referencing an object's state or instance variables. In Smalltalk and Actor, you simply cannot reference an object's internal state (that is, what in Object Pascal we would call its fields) directly. This makes bug propagation a one-way street down along the object hierarchy, which is better than nothing. Unfortunately, even this protection is missing from C++ and the various dialects of Object Pascal. Both Turbo Pascal and QuickPascal require that you, the programmer, enforce encapsulation, and if you're a bad enforcer, you're no better off than with traditional structured programming techniques.

Rubber Truth #3: OOP Allows Easy Free-form Prototyping, and You Can Keep The Prototype!

This is one of those deceptive and infuriating assertions that in certain circumstances might well be right — but for all the wrong reasons. I'm reminded

(continued from page 140)
of a pre-Lego construction toy I had when I was six called "Lock-A-Blox." It was a boxful of brightly-colored blocks, with slots on two faces and tabs on the two opposite faces. All the tabs fit all the slots, and all block dimensions were always even multiples of the smallest. Everything always fit together, no matter how you tried it. There were no wrong combinations, no embarrassing chinks or bulges where things didn't quite fit.

Nonetheless, no matter what you built with it, from the Brooklyn Bridge to the Eiffel Tower to Godzilla, your creation always looked like . . . a bunch of Lock-A-Blox.

Rapid free-form prototyping is indeed possible in Smalltalk and Actor. It's done by taking instances of the hundred-plus object types in the standard class library and hooking them together until things work. You stick menu B behind radio button A, and if they don't look quite right, well, pull them apart and try something else. There's no sense of loss in throwing away the menu because you didn't have to write the code to create the menu. It was all there in the class library, like bubble-packed cold meat hanging on pegs in the refrigerator case down at Safeway.

You can very quickly get a series of menus and other screen controls together in Smalltalk or Actor, and if the prototype user interface isn't too messy, you have a reasonable chance of filling in the guts without having to abandon and rewrite the user interface.

There's a price to be paid: Your application will be made out of pieces written by somebody else, and it will have a look and overall structure dictated by the shape of those pieces. This is not necessarily a bad thing, especially if you're sick of messing around with your own window manager code and you are looking for a standard user interface.

That, however, is for Smalltalk and Actor. There is currently no massive class library in QuickPascal, Turbo Pascal, or Zortech C++. In Object Pascal and C++ you have to build your own Lock-A-Blox, with all the time, effort, and torn hair that that implies.

Duntemann's OOP Warning #3: The Quality of an OOP Prototype Depends Utterly on the Quality of the Standard Class Library

The bottom line of this rubber truth is that the ease of prototyping has less to do with OOP than with a well-thought-out library of prefab program components. I've done some amazingly easy prototyping with Turbo Power Software's Turbo Professional 5.0, and with a superb Modula-2 library product called "Repertoire," all without object-oriented anything. OOP contributes a little — objects are more independent than traditional procedures and functions, and thus, they combine better and with fewer possible side effects — but far less than most people think.

Turbo Pascal prototyping will be made lots easier once Turbo Power's Object Professional library is finished and shipping. (This may well be the case now, while you're reading this; though, while I'm writing, they still have a ways to go.) And over time, such ambitious class libraries will become de rigueur in all OOP languages, just as they have been in Smalltalk and Actor from day one.

Object-oriented programming is no magic pill. It does not allow effortless programming of complex applications that work the first time and never break. It does not make applications run faster, but if badly used can make them run slower. (In fairness to OOP, this is also true of most every programming technique you could name.) Its benefits fall mostly to the programmer: Greater richness of expression in program design; greater potential to reuse well-designed program components; and greater ability to fix, change, and extend existing applications in a maintainable manner. From the outside, the programs will look exactly the same.

The Parent Trap

I treated polymorphism in detail back in November, but it may be time to come back and shed some light into some corners that have remained dark through it all. I lurk on CompuServe a lot, tallying the subjects people have trouble understanding — that's an excellent place for professional explainers, like me, to get their raw material. And whereas the broad concepts of polymorphism seem to be coming across, the details keep getting scrambled and misconstrued.

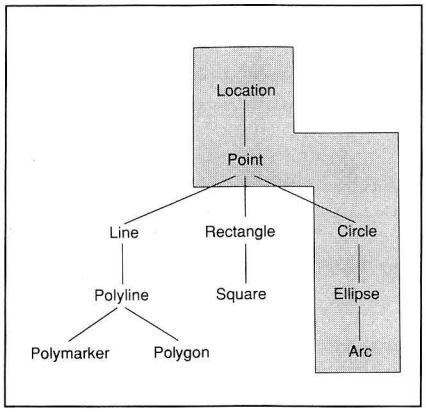


Figure 1: *The domain of object class* ARC

Chief among these is the matter of polymorphic assignment compatibility. In an object hierarchy, assignment compatibility is extended along a given object class' domain. This domain includes the class and all of its descendant classes, as shown in Figure 1. The shaded classes are the domain of class *Arc*.

The rule is this: An object can be assigned to any type "rootward" in its domain. (Let's not argue about "above" or "below," because those words depend utterly on how you draw the hierarchy. In Figure 1, the root of the hierarchy is at the top of the page.) In other words, you could take an instance of type *Arc* and assign it to an instance of type Circle:

```
MyCircle := MyArc;
```

Ordinarily, Pascal would call you down on type conflict grounds, and it still will, if you were to assign *MyArc* to something outside of its domain; say, to an instance of type *Rectangle* or *Polygon*.

One thing to remember that people too often forget is that the reverse is not true: You cannot assign a parent to a child. This won't compile:

```
MyArc := MyCircle;
```

Try to remember this key phrase: "To the general assign the specific." One way to think of an object hierarchy is as a movement from the most general abstract classes at the root to the most specific classes at the leaves. If you'd prefer an earthier mnemonic, try this: "You can hang a leaf on a root. You can't hang a root on a leaf." Think of assigning the specific to the general as hanging a leaf on the root.

Making the Compiler Swallow Polymorphism

The whole idea behind extending assignment compatibility rootward through an object's domain is to make polymorphism syntactically kosher to the compiler. In a sense, polymorphism is the hiding of different classes behind the mask of a common ancestor class. The mask hides the differences between the classes, and emphasizes those aspects the classes have in common — in this case, the aspects defined in the common ancestor class.

In Figure 1, for example, *Line*, *Rectangle*, and *Circle* are all descended from *Point*. *Point* is literally all that the three have in common. Inside *Point* there is a pair of methods for display-

(continued from page 142)
ing and erasing the object's graphic image: *Show* and *Hide*. Inheritance causes *Show* and *Hide* to be passed down into all of *Point*'s child classes. The child classes, however, have the option of redefining their own specific methods under the names *Show* and *Hide*. A line is drawn differently from a circle, so *Line.Show* would necessarily be different from *Circle.Show*.

Nonetheless, the method name and the general idea (to display a figure on the screen) is the same for all the graphic object classes. So bring on the masks:

```
VAR
Figures : ARRAY[1..3] OF Point;
ACircle : Circle;
ARectangle : Rectangle;
ALine : Line;

. . .

Figures[1] := ACircle;
Figures[2] := ALine;
Figures[3] := ARectangle;

FOR I := 1 TO 3 DO
   Figures[I].Show;
```

Here we have an array of type *Point*, to which we assign instances of *Circle*, *Line*, and *Rectangle* as we choose. Because *Point* is within the domain of all three, extended assignment compatibility allows the compiler to accept these assignments, even though they violate traditional Pascal strong typing.

Polymorphism enters the picture in the *FOR* statement, where we step through the array, calling the *Show* method belonging to each array element. Even though the elements of the array *Figures* are all nominally type *Point*, late binding and polymorphism allow the *Circle.Show* method to be invoked for element 1, the *Line.Show* method for element 2, and the *Rectangle.Show* method for element 3. Without extended assignment compatibility, the three graphics figure objects could never have been assigned to the array, and the polymorphic method calls could never have happened.

It's just that the hardest work gets done up front, and the rest of it seems easy only by comparison

The Parting of the Pascals

One (serious) catch. The syntax shown before doesn't work in Turbo Pascal. It compiles correctly, but polymorphism won't happen. In other words, if you compile the above code fragment under Turbo Pascal, the *FOR* statement will execute *Point.Show* for all three of its array elements — as if the polymorphic assignments to the three descendent classes never took place.

This isn't a bug — it's part of the spec. QuickPascal will compile and run code like the example shown earlier. The two Object Pascals differ significantly in their object philosophy (while agreeing in almost everything else), but the most fundamental difference between them is that all QuickPascal objects are allocated on the heap, whereas Turbo Pascal objects may be allocated either on the heap or (by default) in the data segment. Turbo Pascal objects allocated in the data segment (static objects) and addressed directly cannot take part in polymorphism.

Keep this in mind, always: Polymorphism is by nature a dynamic business, and is done through pointers. If an object is not accessed through a pointer, it won't polymorph.

So how, then, does QuickPascal manage to make the example code above run correctly? It's an aspect of the between-two-worlds nature of objects in QuickPascal. When a QuickPascal object is declared, it's declared statically, just as any variable is:

```
VAR
   ACircle : Circle;
```

However, declaring a QuickPascal object is not enough. Before it can be used (actually, before it really exists) it must be allocated on the heap, via *New*:

```
New(ACircle);
```

Thereafter, the object can be referenced as though it were a static variable:

```
ACircle.Show;
```

No caret symbols here, even though *ACircle* exists entirely on the heap. In a sense, QuickPascal objects are neither fully static nor fully dynamic. They're just . . .well. . . objects, that's all.

Behind the scenes, an apparently static reference to a QuickPascal object really is a pointer reference. The name of a QuickPascal object is in truth a pointer to that object on the heap. They've just done away with the familiar caret or "up arrow" pointer reference notation.

In Turbo Pascal, by contrast, an object is either fully static and exists in the data segment, or is fully dynamic and exists on the heap and must be accessed through the traditional pointer reference notation. A Turbo Pascal object is best approached as a record with some special properties, and obeys all the rules a record would with respect to allocation and referencing.

Thus, when in QuickPascal, we assign object instances to an array of object instances, we're actually assigning object pointers to an array of object pointers:

```
Figures[1] := ACircle;
Figures[2] := ALine;
Figures[3] := ARectangle;
```

Here, *ACircle* is actually a pointer to a block of memory on the heap containing the object's actual data and method table. Similarly, the *Figure*'s array is in reality an array of pointers — not whole objects. If we had intended to use the elements of *Figures* without assigning already-allocated objects to them, we would have had to allocate each individual element of *Figures* with the *New* statement:

```
New(Figures[1]);
New(Figures[1]);
New(Figures[1]);
```

This static/dynamic duality of QuickPascal objects (and Apple Pascal objects before them) is a source of some serious confusion to newcomers. Best

to remember: All QuickPascal objects are at the end of pointer references, whether it looks like they are or not!

Polymorphism Through Pointers

Fortunately, both QuickPascal and Turbo Pascal objects can be manipulated through explicit pointer references. And just as assignment compatibility of objects is loosened up within an object's domain, so is the assignment compatibility of pointers to those objects.

In other words, just as a *Circle* object may be assigned to a *Point* object, a pointer defined as a pointer to *Circle* may be assigned to a pointer defined as a pointer to *Point:*

```
TYPE
   PointPtr = ^Point;
   CirclePtr = ^Circle;

VAR
   CircleHandle : CirclePtr;
   PointHandle : PointPtr;

   . . . .

   Pointhandle := CircleHandle;
```

In this situation, polymorphism will work with either compiler, using identical syntax.

The best way to show polymorphism through pointers is with a complete program. There's no substitute for loading up a piece of code and watching it work before your eyes. Listings One and Two, page 158, are pretty much the same program, implemented as their particular compilers demand. Listing One is the QuickPascal version, and Listing Two is the Turbo Pascal 5.5 version.

The programs implement two object classes: *Father* and *Son. Son* is a descendant of *Father.* Both have methods named *Talk,* which display appropriate text to the screen for each class. *Father* says one thing; *Son* says another. The main body of each program does three things:

1. Allocates and initializes three objects, *Dad* (of type *Father*), *Twerp* (of type *Son*), and *Person* (also of type *Father*).
2. Attempts polymorphism through direct references. This works for QuickPascal and fails for Turbo Pascal.
3. Attempts polymorphism through pointer references. This works identically for both compilers.

Assigning a *Son* instance to a *Father* instance is done in both programs:

```
Person := Twerp;
Person.Talk;
```

The method invocation will call *Son.Talk* in QuickPascal and *Father.Talk* in Turbo Pascal. Why?

Person and *Twerp* are both pointers beneath the skin of QuickPascal, and assigning *Twerp* to *Person* only copies the *Twerp* pointer to the *Person* pointer. This is easy and fast — all pointers are the same size, 32 bits.

In the Turbo Pascal version, on the other hand, *Person* and *Twerp* are both objects in the data segment. They are of different sizes. Assigning *Twerp* to *Person* means moving physical data from one location in the data segment to another. To avoid disrupting data beyond the bounds of *Person, Twerp* will have to be truncated during the move — because *Twerp* is larger than *Person.* Data is lost, which is never a good idea.

If you look beneath the surface of the Turbo Pascal version, you'll find that data is moved from *Twerp* to *Person,* and that the *Twerp.Girlfriend's* field is lost in the move. *Person* retains its identity as a *Father* object, however, and any call to a method in *Person* will invoke *Father*'s methods rather than *Son*'s.

The key here is *Person*'s link to the Virtual Method Table, which I described in some detail in my November column. When *Twerp* is assigned to *Person,* all of *Twerp*'s data is copied over *Person*'s, but the VMT links are not involved in the move. *Person*'s VMT link remains intact, even after the assignment overwrites all of *Person*'s data with *Twerp*'s. Why? The VMT contains the size of its object, returned by the *Sizeof* function and used internally by other aspects of the run-time library. Especially when you're dealing with statically allocated data, you do not want to lose track of how big an individual item is. Get it wrong during something as simple as an assignment, and you overwrite adjacent data.

In short, static data is no place for polymorphic fooling around. It's much safer on the heap, where you can throw pointers around instead of whole objects.

Much better to do it this way:

```
Link := @Dad;
KidLink := @Twerp;
Link := KidLink;
Link^.Talk;
```

Here, *Link* is a pointer to *Father* object *Dad,* and *KidLink* is a pointer to *Son* object *Twerp.* Assigning *KidLink* to *Link* can be done because *Link*'s referent shares *KidLink*'s referent's domain. And this time, specifying *Link^.Talk* executes *Son.Talk* because polymorphism works through pointer references in both QuickPascal and Turbo Pascal. No data is moved; once the assignment happens, *Link* points directly to *Twerp,* so *Link^.Talk* executes *Twerp*'s *Talk* method.

The best way to show polymorphism through pointers is with a complete program

The Limits of Polymorphism

This is terrific stuff — hiding the son behind a mask of the father. There's a catch: The only object methods or data accessible through polymorphic assignment are those that the two involved classes have in common. This is subtle, and causes a lot of head scratching during the learning process.

Consider: We've assigned a pointer to *Twerp* to a pointer to *Dad.* Now, can we access a field specific to *Twerp* through that pointer? No!

Try adding this to either Listing One or to Listing Two after the assignment of *KidLink* to *Link:*

```
Writeln(Link^.Girlfriends);
```

Now, *Link* points to a *Son* object, but the compiler will steadfastly claim ignorance that the *Girlfriends* field exists. (Turbo Pascal will give you error 44, and QuickPascal will give you error 60.) Hell, if we can access *Twerp*'s methods through *Link,* we should be able to access *Twerp*'s fields! And we can — but only those fields that *Twerp* and *Dad* both have in common. Type *Father* defines an *Age* field and *Son* inherits it, but *Son* defines a *Girlfriends* field that *Father* doesn't have. A pointer defined as pointing to a *Father* type (as *Link* is) can't access things (either methods or data) that aren't defined in type *Father.*

Why not? Well, loosening assignment

compatibility rules doesn't mean throwing them away entirely. Neither compiler is quite smart enough to deduce things across an assignment statement. Assigning a *SonPtr* to a *FatherPtr* doesn't change the fact that the *FatherPtr* is a *FatherPtr* in the compiler's symbol table. When you try to access a field called *Girlfriends* through a *FatherPtr*, the compiler checks the fields defined under type *Father* for a field called *Girlfriends* and doesn't see it.

If you want to access *Son*-specific fields through a *FatherPtr* you'll have to help the compiler a little, with some "Pizza Terra Typecasting." (See my May 1989 column for an explanation of that peculiar reference . . . .) Try this in either Listing One or Listing Two, again, after the assignment of *KidLink* to *Link*:

```
Writeln(SonPtr(Link)^.Girlfriends);
```

This time it works, both from a compile-time and a run-time perspective. Casting *Link* (a *FatherPtr*) onto a *SonPtr* lets the compiler check and see that, yes, there really is a *Girlfriends* field associated with the pointer in question. All of the usual dangers of typecasting apply. As always, do what you must; just know what you're doing.

Typecasting used in this way kills the magic of polymorphism, in that what we're doing is peeking behind the *Father* mask to see the *Son* hiding there, and acting on that knowledge. Theoretically, we aren't supposed to know the type of the object "behind" a polymorphic assignment. However, if we want to make use of elements specific to the actual type of the object, peeking is our only recourse.

Typecasting your way around polymorphic type-blindness can get you out of some tight corners on occasion. One way to avoid typecasting is to migrate fields and methods "up" the hierarchy so that all classes within a domain share all fields and methods, even if the fields are not used and the methods are empty in the more general classes. In our simple example, this would mean giving the *Girlfriends* field to *Father* and letting *Son* inherit it, just as is done with the *Age* field. Better still, (and at the risk of sounding a little too Eastern) let the son be a son while he is a *Son*, but when he is playing *Father* let him leave his girlfriends at home . . . .

Products Mentioned

Programmer's Productivity Pack
Falk Data Systems
5322 Rockwood Court
El Paso, TX 79932
915-584-7670
$79.95

Mastering Turbo Pascal 5.5
by Tom Swan
Howard W. Sams & Company, 1989
ISBN 0-672-48450-1
Softcover, 875 pages
$25.95
Listings diskette $20.00

Repertoire
PMI
4536 S.E. 50th
Portland, OR 97206
503-777-8844
DOS: $149
OS/2: $189

Programmer's Productivity Pack

I'll never forget the rush of loading Sidekick for the first time back in 1984, and seeing a calculator that worked in hex. No more fiddling on paper and machine gun bashing on my ancient SR10. Sidekick changed the way I approached certain things in programming, because there was no longer any need to avoid working in pure hex or even binary.

The notion of memory resident tools took off with Sidekick, but mostly toward the users — DOS shells to make picking files easier, things like that. It's nice to find an occasional TSR tool that won't do the users a damned bit of good. Such a one is Falk Data Systems' Programmer's Productivity Pack (PPP).

The PPP is a loose collection of useful gadgetry centering on a world-class programmer's calculator with four memories and a full set of bit-wise logical operators, including AND, OR, NOT, XOR, shifts, and rotates. The binary display field allows you to set individual bits rather than enter 31 digits just to set the MSB to 1. Values are displayed simultaneously in binary, hex, decimal, and (gakkh!) octal.

The calculator is good, but the most innovative gadget in the PPP is a sort of empirical keyboard reference screen that displays the scan codes of any key combination you can press on the keyboard. If you need to look up the scan code for a key combination, don't bother with a book; just press the key combination — and you'll get the scan code. As a bonus, attached to certain key combinations are helpful notes indicating whether certain PC-family BIOS versions do or don't recognize a given key combination. Key code information is also given for the dBase data base and dBase compatible languages such as Clipper and QuickSilver, along with helpful notes such as pointing out that dBase and friends consider the Ctrl-\ and F1 keys to be identical.

The shift and lock keys are also displayed as the bit patterns they really are. Beats thumbing through some tech ref manual any day. Other details are right on: Redefinition of PPP's several hotkey assignments, the ability to unload the utility from memory on command, an unusually readable manual, and a superb paper (!) ASCII code chart that I have framed on the wall beside the machine. There's more to it but I'm out of space at this point. Highly recommended.

Memory Hog Heaven

This is a good time to be a programmer. Tom Swan's newest book, *Mastering Turbo Pascal 5.5* is on the streets; finally something good in print about objects. Eighty nanosecond, 1-Mbyte SIMMS are down to about $125 if you shop; by the time you read this they could be at $100. You can get a 20-MHz 386 system with 4 Mbytes of RAM for about $2500. With Windows/386 you can load Turbo Pascal, QuickPascal, PageMaker, and Paradox each into its own virtual-86 partition and carom from one to another like a rock off a canyon wall. Software development is more than just code; it's also data dictionary management and documentation. With Windows/386, you can build your own dream system for creating that ultimate app.

Good old days? We're here, guys. Really.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 158.)

Vote for your favorite feature/article.
Circle Reader Service **No. 11.**

Listing One *(Text begins on page 124.)*

```
/*************************************************************
**                      ---- LOCATE.C -----
**          Copyright (C) 1989 by Mark R. Nelson
**  Program:  LOCATE.C
**  Author:   Mark R. Nelson
**  Summary:  LOCATE reads in MS-DOS formate EXE files and writes
**            out relocated code in Intel Hex format.  The code
**            segment is relocated to start at F000:0000.  Data
**            is relocated to start at 0040:0000.
*************************************************************/

#include <stdio.h>
#include <stdlib.h>

#define TRUE 1
#define FALSE 0

struct exe_header {
                    unsigned int signature;
                    unsigned int image_length_mod_512;
                    unsigned int file_size_in_pages;
                    unsigned int num_of_relocation_table_items;
                    unsigned int size_of_header_in_paragraphs;
                    unsigned int min_num_of_paragraphs_required;
                    unsigned int max_num_of_paragraphs_required;
                    unsigned int disp_of_stack_in_paragraphs;
                    unsigned int initial_sp;
                    unsigned int word_checksum;
                    unsigned int initial_ip;
                    unsigned int disp_of_code_in_paragraphs;
                    unsigned int disp_of_relocation_table;
                    unsigned int overlay_number;
                  } header;

FILE *exe_file;
FILE *hex_file;
unsigned char *image;
unsigned long int image_size;
unsigned long int image_offset;
unsigned long first_data_segment_in_exe_file;
int verbose=TRUE;
unsigned int output_base_code_segment=0xF000;
unsigned int output_base_data_segment=0x0040;

main(int argc,char *argv[])
{
    printf("Locate 1.0  Copyright (C) 1989 by Mark R. Nelson\n");

    open_files(argc,argv);          /* Open the input and output files      */
    read_header_data();             /* Read in the EXE file header          */
    read_input();                   /* Read the code image into a buffer    */
    process_relocation_table();     /* Relocate all segment references      */
    dump_output();                  /* Write the code image to the HEX file */
    output_restart_code();          /* Write the restart code line          */
    output_intel_hex(0,0,1,NULL);   /* Output an EOF record                 */
}

/*************************************************************
**                      ---- open_files ----
**   This routine opens an EXE file and a HEX file.  If they are specified
**   on the command line, those names are used.  Otherwise the user is
**   prompted for file names
*************************************************************/

open_files(int argc,char *argv[])
{
char exe_file_name[81];
char hex_file_name[81];

    if (argc>1)
        strcpy(exe_file_name,argv[1]);
    else
    {
        printf("EXE file name? ");
        scanf("%s",exe_file_name);
    }

    exe_file=fopen(exe_file_name,"rb");
    if (exe_file==NULL)
        fatal_error("Had trouble opening the input file!");

    if (argc > 2)
        strcpy(hex_file_name,argv[2]);
    else
    {
        printf("Hex file name? ");
        scanf("%s",hex_file_name);
    }

    hex_file=fopen(hex_file_name,"w");
    if (hex_file==NULL)
        fatal_error("Had trouble opening the output file!");

}

/*************************************************************
**                   ---- read_header_data ----
**   This routine reads in the EXE header structure and computes both
**   the image offset and size.  The compuataions are all done using
**   numbers found in the header.  This program arbitrarily limits
**   the code image size to 64K, but could easily be expanded to go
**   to larger sizes.
*************************************************************/

read_header_data()
{
    if (fread(&header,sizeof(struct exe_header),1,exe_file) != 1)
        fatal_error("Couldn't read header from file!");
    if (verbose)
        print_header();
    image_offset=header.size_of_header_in_paragraphs*16;
    image_size = (header.file_size_in_pages-1)*512;
    image_size -= image_offset;
    image_size += header.image_length_mod_512;
    if (image_size > 0xFFFFL)
        fatal_error("The EXE image is larger than I can handle!");
    first_data_segment_in_exe_file=header.disp_of_stack_in_paragraphs;
}

/*************************************************************
**                      ---- read_input --
**   This routine reads the code image into a buffer.  Any trouble with
**   the buffer or the file generates a fatal error.
*************************************************************/

read_input()
{
    image=malloc(image_size);
    if (image==NULL)
        fatal_error("Couldn't allocate output image space!");
    if (fseek(exe_file,image_offset,SEEK_SET) != 0)
        fatal_error("Couldn't seek to image in the input file!");
    if (fread(image,1,(int)image_size,exe_file) != (int)image_size)
        fatal_error("Couldn't read in the image!");
}

/*************************************************************
**                  ---- process_relocation_table ----
**   This routine loops through all of the entries in the relocation
**   table.  Each entry points to a segment value in the code image.
**   That segment value is checked to see if it points to code or
**   data.  If it points to data, it is relocated to start at the
**   output_base_data_segment.  Code segments are relocated to start
**   at output_base_code_segment.
*************************************************************/

process_relocation_table()
{
int i;
unsigned int reloc[2];
unsigned long int spot;
unsigned int *guy;
unsigned int old_value;
unsigned int new_value;

 fseek(exe_file,(long)header.disp_of_relocation_table,0);
 for (i=0;i<header.num_of_relocation_table_items;i++)
 {
  if (fread(reloc,2,2,exe_file) != 2)
    fatal_error("Couldn't read relocation data from file!");
  printf("Record %3d:  %04X:%04X:",i,reloc[1],reloc[0]);
  spot=reloc[1]*16 + reloc[0];
  old_value=*(int *)(image+spot);
  printf("  was: %04X", old_value);
  if (old_value < first_data_segment_in_exe_file)
   new_value=old_value+output_base_code_segment;
  else
   new_value=old_value-
first_data_segment_in_exe_file+output_base_data_segment;
  *(int *)(image+spot)=new_value;
  printf("  now is: %04X\r", new_value);
 }
 printf("\n");
}

/*************************************************************
**                    ---- dump_output ----
**   This routine loops the entire code image.  It outputs 34 bytes
**   at a time in Intel Hex format until it is done.  While it is
**   doing this it keeps the user posted by writing the addresses
**   to the screen.  Note that this module would need some modifications
**   to handle images greater than 64K.
*************************************************************/

dump_output()
{
unsigned char *output_pointer;
long int output_size;
int record_size;
unsigned int output_address;
unsigned char segment_address[2];

    output_pointer=image;
    output_size=image_size;
    output_address=0;
    segment_address[0]=output_base_code_segment>>8;
    segment_address[1]=output_base_code_segment & 0xff;
    output_intel_hex(2,0,2,segment_address);
    while (output_size > 0)
    {
        printf("%04X\r",output_address);
        record_size=(output_size > 34) ? 34 : output_size;
        output_intel_hex(record_size,output_address,0,output_pointer);
        output_pointer += record_size;
        output_size -= record_size;
        output_address += record_size;
    }
    printf("\n");
}

/*************************************************************
**                   ---- output_restart_code ----
**   This routine writes a JMP START instruction out at location
**   at FFFF:0000.  The address of START is contained in the EXE
**   header block.
*************************************************************/

output_restart_code()
{
```

```
unsigned char jmp_code[5];
unsigned char segment_address[2];

    segment_address[0]=0xff;
    segment_address[1]=0xff;
    output_intel_hex(2,0,2,segment_address);
    jmp_code[0]=0xea;                   /*  JMP ????:????           */
    jmp_code[1]=header.initial_ip & 0xff;
    jmp_code[2]=header.initial_ip >> 8;
    header.disp_of_code_in_paragraphs += output_base_code_segment;
    jmp_code[3]=header.disp_of_code_in_paragraphs & 0xff;
    jmp_code[4]=header.disp_of_code_in_paragraphs >> 8;
    header.disp_of_code_in_paragraphs -= output_base_code_segment;
    output_intel_hex(5,0,0,jmp_code);
}

/*********************************************************************
**                  ---- output_intel_hex ----
**   This routine writes a single record of Intel Hex.
*********************************************************************/

output_intel_hex(int size,unsigned int address,int type,unsigned char buffer[])
{
int checksum;
int i;

    fprintf(hex_file,":%02X%04X%02X",size,address,type);
    checksum=size+address+(address>>8)+type;
    for (i=0;i<size;i++)
    {
        fprintf(hex_file,"%02X",buffer[i]);
        checksum += buffer[i];
    }
    checksum = -checksum & 0xff;
    fprintf(hex_file,"%02X\n",checksum);
}

/*********************************************************************
**                    ---- print_header ----
**   This is a routine that lets the program print out the contents
**   of the header.  It is here primarily for assistance in debugging.
*********************************************************************/

print_header()
{
    printf("Link program signature:                                  ");
    printf("%4.4X\n",header.signature);
    printf("Length of image mod 512:                                 ");
    printf("%4.4X\n",header.image_length_mod_512);
    printf("Size of file in 512 byte pages, including header:        ");
    printf("%4.4X\n",header.file_size_in_pages);
    printf("Number of relocation table items:                        ");
    printf("%4.4X\n",header.num_of_relocation_table_items);
    printf("Size of header in 16 byte paragraphs:                    ");
    printf("%4.4X\n",header.size_of_header_in_paragraphs);
    printf("Minimum # of 16 byte paragraphs needed above program:    ");
    printf("%4.4X\n",header.min_num_of_paragraphs_required);
    printf("Maximum # of 16 byte paragraphs needed above program:    ");
    printf("%4.4X\n",header.max_num_of_paragraphs_required);
    printf("Displacement of stack within load module in paragraphs:  ");
    printf("%4.4X\n",header.disp_of_stack_in_paragraphs);
    printf("Offset to be loaded in SP:                               ");
    printf("%4.4X\n",header.initial_sp);
    printf("Word checksum:                                           ");
    printf("%4.4X\n",header.word_checksum);
    printf("Offset to be loaded in IP:                               ");
    printf("%4.4X\n",header.initial_ip);
    printf("Displacement of code segment in 16 byte paragraphs:      ");
    printf("%4.4X\n",header.disp_of_code_in_paragraphs);
    printf("Displacement of 1st relocation table item:               ");
    printf("%4.4X\n",header.disp_of_relocation_table);
    printf("Overlay number:                                          ");
    printf("%4.4X\n",header.overlay_number);
}
/*********************************************************************
**                    ---- fatal_error ----
**   A self-documenting utility.
*********************************************************************/

fatal_error(char *message)
{
    printf(message);
    exit(1);
}
```

End Listing One

Listing Two

```
;*********************************************************************
;                       ---- START.ASM ----
;           Copyright (C) 1989 by Mark R. Nelson
;  Module:   START.ASM
;  Author:   Mark R. Nelson
;  Summary:  This module is an alternate startup routine for Microsoft
;            or Turbo C programs running on non DOS hardware.  It has
;            three main jobs.  First, it sets up the segment definitions
;            so that the STACK segment is the first segment in RAM.
;            Second, it initializes all predefined data.  Third, it jumps
;            to the user's main() routine.
;*********************************************************************

;
;   Note here that if DGROUP does not contain the stack, which may
;   be true for larger models, the STACK segment needs to be moved
;   to be the first one after END_OF_ROM.
;
;   Also note that for the startup code to work properly, the first
;   segment in the data area must be paragraph aligned.  This insures
;   that for the startup code, the first data segment is exactly 3
;   larger than the END_OF_ROM segment.
;
_TEXT               SEGMENT BYTE PUBLIC 'CODE'
_TEXT               ENDS
END_OF_ROM          SEGMENT PARA PUBLIC 'STARTUP_CODE'
END_OF_ROM          ENDS
_CONST              SEGMENT PARA PUBLIC 'CONST'
_CONST              ENDS
_BSS                SEGMENT WORD PUBLIC 'BSS'
_BSS                ENDS
_DATA               SEGMENT WORD PUBLIC 'DATA'
_DATA               ENDS
_STACK              SEGMENT WORD STACK 'STACK'

MYSTACK             DB 512 DUP (?)

_STACK              ENDS
;
;   Note here that if DGROUP does not contain the stack, which may
;   be true for larger models, the STACK segment needs to be moved
;   to be the first one after END_OF_ROM.
;
DGROUP              GROUP   _CONST,_BSS,_DATA,_STACK

                    extrn   _main:far

                    public  __acrtused          ;This value makes many of the
__acrtused = 9876h                              ;library routines happy

END_OF_ROM SEGMENT PARA PUBLIC 'STARTUP_CODE'

                    ASSUME  CS:END_OF_ROM

                    PUBLIC  START

START               PROC    FAR

                    MOV     AX,DGROUP           ;This code initializes the
                    MOV     SS,AX               ;SS:SP pair with the proper
                    MOV     SP,OFFSET MYSTACK+512 ;values.
;
;   This section of code is charged with moving all predefined values out
;   of ROM and into RAM.  This is done by copying all values out of the
;   part of ROM immediately following the last code segment into the
;   data section.
;
                    MOV     AX,CS           ;The present code segment is the
                    ADD     AX,3            ;last code segment, and we know
                    MOV     DS,AX           ;that the first data segment will
                                            ;be three up form here.  Put that
                                            ;value into DS

                    MOV     AX,DGROUP       ;Now set ES to point to the first
                    MOV     ES,AX           ;section of RAM.
                    XOR     SI,SI           ;SI and DI are the registers
                    XOR     DI,DI           ;used in the MOVSB instruction.
                    MOV     CX,0FBFFH       ;This rep instruction will fill
                    REP     MOVSB           ;everything in a 64K RAM following
                                            ;the interrupt vector space.

                    MOV     AX,ES           ;Now set up DS to point to DGROUP
                    MOV     DS,AX
                    STI                     ;Enable interrupts and the jump
                    JMP     _main           ;to the start of the C code

START               ENDP

END_OF_ROM          ENDS

                    END     START
```

End Listing Two

Listing Three

```
/*********************************************************************
**                      ---- HELLO.C -----
**           Copyright (C) 1989 by Mark R. Nelson
**  Program:  HELLO.C
**  Author:   Mark R. Nelson
**  Summary:  Hello demonstrates the LOCATE program.  It simulates
**            output of a string to a printer on an embedded system.
*********************************************************************/

main()
{

    my_print("Hello, world!\n");
    while (1) ;
}

my_print(char *message)
{
    while (*message)
    {
        if (*message=='\n')
        {
            while ((inportb(0x200) & 0x80) == 0) ;
            outportb(0x200,'\r');
        }
        while ((inportb(0x200) & 0x80) == 0) ;
        outportb(0x200,*message++);
    }
}
```

End Listings

Listing One *(Text begins on page 132.)*

```
/* ---------- textsrch.h ---------- */

#define OK    0
#define ERROR !OK

#define MXTOKS 25           /* maximum number of tokens */
#define MAXFILES 512        /* maximum number of files  */
#define MAPSIZE MAXFILES/16 /* number of ints/map       */

/* ---- the search decision bitmap (one bit per file) ---- */
struct bitmap {
    int map[MAPSIZE];
};

/* ------- the postfix expression structure -------- */
struct postfix  {
    char pfix;      /* tokens in postfix notation */
    char *pfixop;   /* operand strings            */
};

/* --------- the postfix stack ---------- */
extern struct postfix pftokens[];
extern int xp_offset;

/* --------- expression token values ---------- */
#define TERM     0
#define OPERAND 'O'
#define AND     '&'
#define OR      '|'
#define OPEN    '('
#define CLOSE   ')'
#define NOT     '!'
#define QUOTE   '"'

/* ---------- textsrch prototypes ---------- */
struct postfix *lexical_scan(char *expr);
struct bitmap exinterp(void);
void init_database(void);
struct bitmap search(char *word);
void process_result(struct bitmap);
```

End Listing One

Listing Two

```
/* ------------ exinterp.c --------------- */

/*
 * An expression interpreter that processes the
 * tokens on a postfix stack.
 */

#include "textsrch.h"

static struct postfix *pf = pftokens;

static struct bitmap pop(void);
static struct bitmap not(struct bitmap map1);
static struct bitmap and(struct bitmap map1, struct bitmap map2);
static struct bitmap or(struct bitmap map1, struct bitmap map2);

/* ----- entry to the interpreter
        returns a bitmap which indicates the files
        that match a search expression ------------- */
struct bitmap exinterp(void)
{
    /* ------- find the top of the postfix stack ------- */
    while (pf->pfix != TERM)
        pf++;
    /* ------- get the result of the expression ------- */
    return pop();
}

/* ------ pops an operand and converts it to a bit map ------ */
static struct bitmap pop(void)
{
    struct bitmap map1;
    switch ((--pf)->pfix)   {
        case OPERAND:
            map1 = search(pf->pfixop);
            break;
        case NOT:
            map1 = not(pop());
            break;
        case AND:
            map1 = and(pop(), pop());
            break;
        case OR:
            map1 = or(pop(), pop());
            break;
        default:
            break;
    }
    return map1;
}

/* ------- unary <not> operator ----------- */
static struct bitmap not(struct bitmap map1)
{
    int i;
    for (i = 0; i < MAPSIZE; i++)
        map1.map[i] = ~map1.map[i];
    return map1;
}
```

```
/* ------- binary <and> operator -------------- */
static struct bitmap and(struct bitmap map1, struct bitmap map2)
{
    int i;
    for (i = 0; i < MAPSIZE; i++)
        map1.map[i] &= map2.map[i];
    return map1;
}

/* ------- binary <or> operator -------------- */
static struct bitmap or(struct bitmap map1, struct bitmap map2)
{
    int i;
    for (i = 0; i < MAPSIZE; i++)
        map1.map[i] |= map2.map[i];
    return map1;
}
```

End Listing Two

Listing Three

```
/* ---------- search.c ----------- */

/*
 * stub functions to simulate the TEXTSRCH retrieval process
 */

#include <stdio.h>
#include <dir.h>
#include <string.h>
#include <process.h>
#include "textsrch.h"

static char names[MAXFILES][13];
static int namect;
static void setbit(struct bitmap *map1, int bit);
static int getbit(struct bitmap *map1, int bit);

/* --------- initialize the data base environment --------- */
void init_database(void)
{
    struct ffblk ff;
    int rtn;
    rtn = findfirst("*.txt", &ff, 0);
    while (rtn != -1 && namect < MAXFILES)  {
        strcpy(names[namect++], ff.ff_name);
        rtn = findnext(&ff);
    }
}
/* ------ search the data base for a match on a word ------ */
struct bitmap search(char *word)
{
    int i;
    FILE *fp;
    char str[80];
    struct bitmap map1;

    for (i = 0; i < MAPSIZE; i++)
        map1.map[i] = 0;
    sprintf(str, "grep -l -i %s *.txt >hits", word);
    system(str);
    if ((fp = fopen("hits", "rt")) != NULL) {
        while (fgets(str, 80, fp) != NULL)  {
            *strchr(str, ':') = '\0';
            for (i = 0; i < MAXFILES; i ++) {
                if (stricmp(str+5, names[i]) == 0)  {
                    setbit(&map1, i);
                    break;
                }
            }
        }
        fclose(fp);
    }
    return map1;
}

/* ---- process the result of a query expression search ---- */
void process_result(struct bitmap map1)
{
    int i;
    for (i = 0; i < MAXFILES; i++)
        if (getbit(&map1, i))
            printf("\n%s", names[i]);
}

/* ------ sets a designated bit in the bit map ------ */
static void setbit(struct bitmap *map1, int bit)
{
    int off = bit / 16;
    int mask = 1 << (bit % 16);
    map1->map[off] |= mask;
}

/* ------ tests a designated bit in the bit map ------ */
static int getbit(struct bitmap *map1, int bit)
{
    int off = bit / 16;
    int mask = 1 << (bit % 16);
    return map1->map[off] & mask;
}
```

End Listing Three

(Listings continued on page 156)

Listing Four *(Listing continued, text begins on page 132.)*

```
/* ---------- textsrch.c ------------ */

/*
 * The TEXTSRCH program.
 */

#include <stdio.h>
#include <process.h>
#include <string.h>
#include "textsrch.h"

void main(void)
{
    char expr[80];
    /* -------- initialize the text data base --------- */
    init_database();
    do  {
        /* ----- read the expression from the console ------ */
        printf("\nEnter the search expression:\n");
        gets(expr);
        if (*expr)  {
            /* --- scan for errors and convert to postfix --- */
            if (lexical_scan(expr) == NULL) {
                /* ----- the expression is invalid ---- */
                while(xp_offset--)
                    putchar(' ');
                putchar('^');
                printf("\nSyntax Error");
                exit(1);
            }
            /* --- interpret the expression, search the data base,
                and process the hits ------- */
            process_result(exinterp());
        }
    } while (*expr);
}
```

End Listings

Listing One *(Text begins on page 140.)*

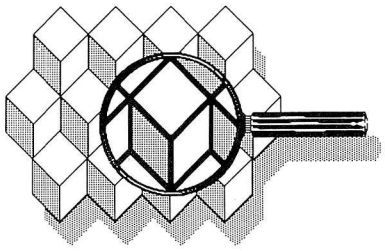
```
PROGRAM PolyTest;  { For QuickPascal 1.0 }

                   { BY Jeff Duntemann   }
                   { For DDJ 1/90        }

USES Crt;

TYPE
  FatherPtr = ^Father;
  Father = OBJECT
             Age : Integer;
             PROCEDURE Init;
             PROCEDURE Talk;
           END;

  SonPtr = ^Son;
  Son    = OBJECT(Father)
             Girlfriends : Integer;
             PROCEDURE Init; OVERRIDE;
             PROCEDURE Talk; OVERRIDE;
           END;


VAR
  Dad     : Father;
  Twerp   : Son;
  Person  : Father;
  Link    : FatherPtr;
  KidLink : SonPtr;


PROCEDURE Father.Init;

BEGIN
  Self.Age := 42;
END;


PROCEDURE Father.Talk;

BEGIN
  Writeln('This is the old man talking.');
END;



PROCEDURE Son.Init;

BEGIN
  Self.Age := 11;
  Self.Girlfriends := 5;
END;


PROCEDURE Son.Talk;

BEGIN
  Writeln('Cool it, daddy-o!');
END;



BEGIN
  ClrScr;

  New(Dad);  { These three statements allocate the objects on the heap }
  New(Twerp);
  New(Person);

  Dad.Init;        { Type Father }
  Twerp.Init;      { Type Son }
  Person.Init;     { Type Father }

  { The following works polymorphically in QuickPascal but not Turbo: }

  Person := Twerp;   { Assign a Son object to a Father object }
  Person.Talk;       { Polymorphism allows the Son.Talk method to be }
                     { executed from Person, which is type Father }
  Writeln('Person''s age is: ',Person.Age);  { Check the age on-screen! }
  Readln;

  { Working through pointers works with either Turbo or Quick Pascal: }

  Link := @Dad;      { Link is defined as a ^Father }
  Link^.Talk;        { Dad speaks here... }

  KidLink := @Twerp; { Twerp is type Son }
  Link := KidLink;   { Assign a SonPtr to a FatherPtr }
  Link^.Talk;        { Even tho Link is a FatherPtr, the kid talks here. }
  Readln;

END.
```

End Listing One

Listing Two

```
PROGRAM PolyTest;   { For Turbo Pascal 5.5 }

                    { BY Jeff Duntemann    }
                    { For DDJ 1/90         }


USES Crt;

TYPE
  FatherPtr = ^Father;
  Father = OBJECT
             Age : Integer;
             CONSTRUCTOR Init;
             PROCEDURE Talk; VIRTUAL;
           END;

  SonPtr = ^Son;
  Son    = OBJECT(Father)
             Girlfriends : Integer;
             CONSTRUCTOR Init;
             PROCEDURE Talk; VIRTUAL;
           END;


VAR
  Dad     : Father;
  Twerp   : Son;
  Person  : Father;
  Link    : FatherPtr;
  KidLink : SonPtr;


CONSTRUCTOR Father.Init;

BEGIN
  Age := 42;
END;


PROCEDURE Father.Talk;

BEGIN
  Writeln('This is the old man talking.');
END;



CONSTRUCTOR Son.Init;

BEGIN
  Age := 11;
  Girlfriends := 5;
END;


PROCEDURE Son.Talk;

BEGIN
  Writeln('Cool it, daddy-o!');
END;



BEGIN
  ClrScr;

  Dad.Init;        { Type Father }
  Twerp.Init;      { Type Son }
  Person.Init;     { Type Father }


  { The following does not work polymorphically in Turbo Pascal: }

  Person := Twerp;  { The assignment is kosher to the compiler...}
  Person.Talk;      { ...but polymorphism does NOT work here...  }
                    { ...and Dad speaks instead of the kid.      }
  Writeln('Person''s age is: ',Person.Age);
  Readln;

  { This, on the other hand, works polymorphically as it should: }

  Link := @Dad;       { Link is defined as a FatherPtr }
  Link^.Talk;         { Dad speaks here... }

  KidLink := @Twerp;  { Twerp is type Son }
  Link := KidLink;    { Assign a SonPtr to FatherPtr }
  Link^.Talk;         { Even tho Link is a FatherPtr, the kid talks here. }
  Readln;

END.
```

End Listings



Overlay Architect and Overlay Optimizer, two tools that manage overlay structures, have been released by **AtLast! Software**. According to AtLast!, Overlay Architect can handle more than 4000 modules and 16,000 code symbols, supports libraries and object modules, works with most compilers, produces nested overlays and multiple overlay areas, supports indirect calls, detects unused functions and modules, supports multiple entry points, as well as full use of the Plink86plus TRACK command. The company claims that this will result in the smallest possible executable program size by using nested overlays and multiple overlay areas; on a 386, a program consisting of a megabyte of code in a thousand modules can be analyzed by Overlay Architect in less than three minutes.

Overlay Optimizer analyzes the performance of a program's overlay structure and determines how overlays should be rebuilt for optimal performance. It requires a Plink86plus map file and the debug output of the sample debug run of a program; you specify the size to which your program can grow, and Overlay Optimizer creates a link response file that reflects the new overlay structure. It is a two-step process: You first have to run your program with the debugging version of the Plink86plus overlay loader, and then run Overlay Optimizer. The Architect and Optimizer can be purchased together for $569, or separately for $369 and $269, respectively. Reader service no. 20.
AtLast! Software, Inc.
449 Mountain View Road
Boulder, CO 80302
303-938-1210

A monitor called the "DB-51 Enhanced System Monitor," for 8051/31-and 80515/535-based single board computers, has been released by **Allen Systems**. This monitor allows for development and debugging of user software right on the target 8051 hardware. It supports a command set to facilitate user operation. The monitor is EPROM-resident, and can be directly installed into the user hardware. After object code output is downloaded from the host into RAM on the target system under DB-51 control, the host is configured as a terminal that allows the user direct access to the DB-51 and its available commands for debugging and testing.

In addition to the command set, DB-51 also features the routing of interrupt service routines through external RAM, and line-oriented command entry which allows simple command-line editing prior to command processing. In addition, DB-51 display information can be suspended, resumed, and aborted. And upon initialization, the DB-51 checks the target hardware and then configures itself for optimum resource utilization.

In addition to the software and user's manual, a listing of the DB-51 source is also available on hardcopy and on disk. It sells for $100. Reader service no. 21.
Allen Systems
2346 Brandon Rd.
Columbus, OH 43221
614-488-7122

Tronix has released a Unix/Xenix Kernel Debugger (KDB). The symbolic debugger runs on SCO's System V/386 and Xenix/386, Interactive Systems' 386/IX, Everex's Enix, and AT&T's System V, Release 3.2. According to Benjamin Chou, president, it is "designed to let the Unix/Xenix system software engineer easily control the execution and environment of software within the Unix/Xenix operating system." The Tronix Kernel Debugger enables programmers to set break point at data address(es), trace any process's stack, display data in a predefined data structure format, display and modify code and data, single step through code, display information about all the processes that are running, and display and modify registers, all within the Unix/Xenix kernel.

Tronix KDB requires 150K-bytes from hard disk during installation, and afterwards the new kernel's size is about 120K-bytes larger than a regular kernel. The product sells for $475, and source code and a free demonstration disk are also available. Reader service no. 22.
Tronix International
10601 S. DeAnza Blvd., Ste. 216
Cupertino, CA 95014
408-973-8559

Microsystems Software has announced CodeRunneR, a highly-optimized library for creating fast, compact TSR programs with full DOS access, using Borland's Turbo C and Microsoft C. Users can supposedly create C programs on a performance and code size par with assembly language at a fraction of the development time and cost. Developers can, says president J. Scott Benson, "stay with high-level languages like C, and yet produce very tight code that can still fit their user's systems."

Among CodeRunneR's features are the elimination of initialization code and data when programs go resident, a function-level granular and auto-initializing run-time library, the ability to create multitasking programs, a BCD floating-point package, full hotkey support, transparent DOS access, and two levels of event schedulers. CodeRunneR lists for $149, and Microsystems offers a 30-day money-back guarantee if you can find a commercially-available C library that delivers smaller or faster run-time code. Reader service no. 23.
Microsystems Software, Inc.
600 Worcester Rd.
Framingham, MA 01701
508-626-8511

Andrews Data Systems has announced the release of object libraries for Borland Turbo Pascal 5.5 and Microsoft QuickPascal 1.0. Both products include a window object library for text and graphics, a remote object support library for distributed processing, and a concurrent object support library for multitasking within applications. The remote object library is bundled with a serial object library that may also be purchased separately.

A network library that is scheduled to be released in December will also support the distributed processing provided under the remote object library using NetBIOS. A Novell implementation is slated for early this year, and future products will include C++ object libraries. Source code is available with all products. The libraries cost $59 each, $89 with source, or $149/$189 for Windows/Remote/Concurrent combination. Both compiler versions are included in each library. Reader service no. 27.
Andrews Data Systems
P.O. Box 37123
Denver, CO 80237
800-255-5550 ext. 615

MetaWare has released High C compiler Version 1.6 for MS-DOS and 386/486DOS. The package includes MetaWare's High C compiler for OS/2 and real-mode MS-DOS. Version 1.6 features expanded libraries, new documentation, two editors, a disk cache utility, a b-tree library, and a graphics library for the 386/486 in protected mode. MetaWare's new *make* facility is included, as well as a set of Unix-style utilities for the MS-DOS operating system. This upgrade comes with the

(continued on page 165)

(continued from page 161)
GFX/386 Graphics library, produced in conjunction with C Source, which provides specific floating-point graphics functions. High C also includes the EC editor from C Source, HyperDisk disk cache from HyperWare, and source code for the MicroEMACS editor.

The High C library is ANSI conformant, and additional library functions bring it to 86 percent compatibility with Microsoft's C libraries. The High C compiler also provides cross-language calling, diagnostics, and configurability. Toggles and pragmas allow developers to select from a variety of compiler features. The High C compiler is discussed in the two-part article "Stalking General Protection Faults" by Andrew Schulman, which begins this month in *DDJ*.

Current licensees covered by the technical support program can upgrade for free. Users of Version 1.5 who do not have this policy can upgrade for $75 on the 8088, 80186, and 80286; the upgrade for the 80386 is $150. Otherwise, the product is licensed for $595 for DOS and $895 for 386/486DOS. Reader service no. 25.
MetaWare Inc.
2161 Delaware Avenue
Santa Cruz, CA 95060-5706
408-429-6382

Sun Microsystems has introduced GUIDE (Graphical User Interface Design Editor), a design tool for building an Open Look graphical user interface for applications. The GUIDE prototyper is an interactive tool that features a palette of icons that represent various objects, such as window control functions, scrollbars, and menus, so a developer can drop an icon into the desired location instead of writing new code. GUIDE automatically generates the user interface code, which can shorten development time. And applications designed with it can offer "drag and drop" file loading, so users can select an icon and the application will automatically load.

The GUIDE prototyper runs on Release 4.X of the SunOS operating system on the X11/NeWS window system. GUIDE media, documentation, and a right-to-use license will be sold unbundled for $250, and will be available sometime during the first quarter of 1990. Reader service no. 26.
Sun Microsystems, Inc.
2550 Garcia Ave.
Mountain View, CA 94043
415-336-6536

Invention Software has released the Extender DialogHandler, the latest addition to their development series. DialogHandler includes over 160 routines which decrease the time required to program modal dialogs with complete functionality.

DialogHandler supports the list manager, and builds in features like range checking, on-the-fly character filtering to maintain number integrity, default item bold outlines, and cut, copy, and paste support with context checking. DialogHandler also supports user hooks, key equivalents, and animated icons and pictures. It comes with complete documentation and 5000 lines of example code, and you can call for free technical support. The Extender DialogHandler with full source code costs $189.95, without $99.95. Reader service no. 28.
Invention Software
P.O. Box 3168
Ann Arbor, MI 48106
313-996-8108

The Software Organization announced a new programming tool, DialogCoder, that is supposed to eliminate 95 percent of the coding normally associated with dialog box programming. The company claims that DialogCoder automatically generates compilable C source code from dialog templates to manage all controls in the dialog; that it uses graphical metaphors to express the relationships between dialog controls and actions; that it allows users to interactively specify the state of each dialog control during initialization and command processing; that it supports listbox initialization from ASCII files, resources bound to an application, and directory lists; that it supports multiline edit initialization from ASCII file; and that it provides validation code for text and number edit controls.

The DialogCoder also supports all controls provided by the Microsoft and Whitewater dialog editors, as well as the entrance and exit processing for edit fields. DialogCoder is designed for both novice and experienced Windows programmers — the learning curve is about one hour for an experienced Windows programmer — and even nonprogrammers can use it, provided they're given detailed dialog design specifications to follow. DialogCoder requires a 286- or 386-based machine with minimum memory for Windows 2.X, and a Microsoft-compatible mouse is optional. The cost is $349. Reader service no. 29.
The Software Organization, Inc.
617-354-2012
800-696-2012

Software Translations Inc. (STI) has announced the release of v7.5 of its B-Tran Basic to C translator, which trans-

lates Microsoft QuickBasic v4.5 and the recently announced Basic 7.0 to C source code. Software Translations guarantees 99 percent translation of code, including user-defined variables, multidimensional dynamic arrays, and named COMMON blocks. B-Tran is available for all machines running DOS, Unix, Xenix, AIX, Ultrix, and VMS. It translates more than 1000 lines per minute of QuickBasic code into C that complies with SVID, X/Open, ANSI, and K&R standards. One command translates, compiles, and links. The C code it produces is supposedly readable for Basic developers, though it does offer the appearance of a professional C program. And B-Tran translated code has none of the memory or code size restrictions of QuickBASIC. Pricing starts at $449 for the Microsoft C compiler under DOS. Other translators available include DEC VAX Basic, DEC Basic + and Basic +2, TI Basic, and CBasic. Technical support and maintenance is also provided. Reader service no. 30.
Software Translations Inc.
The Carriage House
28 Green St.
Newburyport, MA 01950
508-462-5523

The NetWare Programmer's Workbench, a collection of software development tools for client/server applications in a networked environment, has been announced by **Novell**. The product includes the Novell/Watcom C Network Compiler for developing client-side applications and the C Network Compiler/386 for developing server-side applications. Also included are the NetWare RPC (remote procedure call), the Phar Lap 386 assembler, a library of functions that are ANSI C and IEEE POSIX-compliant, and a prerelease version of NetWare 386 v3.1 SDK.

According to Nancy Woodward, Novell's vice president and general manager of development products, "The NetWare Programmer's Workbench allows programmers to build distributed applications using standard programming tools and familiar procedures." Novell intends for this factor to increase the rate at which these new applications are developed. And programmers are not limited to the NetWare APIs — you can now design your own for writing to the network operating system and install them as NLMs (NetWare Loadable Modules), which are dynamically linkable modules that allow the network operating system functions to be easily extended.

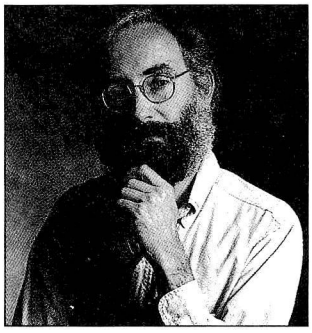
The NetWare Programmer's Workbench is available to qualified Strategic Partners for $3995. The C Network Compiler/386 and C Network Compiler are available for $995 and $695, respectively. Reader service no. 40.
Novell Development Products
P.O. Box 9802
Austin, TX 78766
512-346-8380

NEMO 2.0, a real-time, rule-based expert system development package written in C, is available from **S2O Developpement**, a French company. NEMO 2.0 has the ability to monitor an ongoing process and respond to events as they occur, through its "Perception Module." NEMO 2.0 can be applied for maintenance functions and safety evaluations, and provides real-time capabilities for applications in areas such as telecommunications and the petroleum industry.

NEMO 2.0 consists of a facts base, a rule base, and an inference engine. The product development environment includes a knowledge compiler and optimizer for response time; a graphic multi-window developer interface that allows the display of the object hierarchy tree, control of the inference engine, and online help screens; and a graphic multi-window operator interface can be customized to generate windows for facts and explanation display and for creation and animation of process graphical views and synoptics.

NEMO 2.0 runs on a 386 workstation under Unix, on Sun 3, Sun 4, SPARCstation, on HP 9000 Series and on a DEC VAXstation with Ultrix or VMS. 1 Mbyte of RAM and 3 Mbytes on hard disk are required. Reader service no. 41.
Expert Knowledge
1801 Avenue of the Stars, Ste. 507
Los Angeles, CA 90067
213-556-1628

Standing in the Rubble, Looking Up



On October 17, 1989, a major earthquake shook the San Francisco Bay Area, wreaking billions of dollars in property damage and an immeasurable cost in human suffering.

A friend of mine, John Anderson, died in the quake.

Remarkably few lives were lost, considering the magnitude and duration and the extent of property damage. We who live in the Bay Area kept reminding ourselves after the event how lucky we were, but for those of us who had lost a friend, the general good fortune only made our personal loss harder to accept. We felt betrayed by the law of averages, and could only repeat, meaninglessly, that John had been in the wrong place at the wrong time.

I believe that John was in the right place at the right time: It was the earth that was at fault. John had been visiting a software company in San Francisco that Tuesday afternoon, talking about a variety of topics, all of which were really one grand topic: Pushing the envelope. It was John's theme, what he had got into computers for; probably what he had become a writer for and what he was pursuing in his music. He believed, wholeheartedly and without embarrassment, in pushing back the limits to human achievement. The fact that he was pursuing this goal on that afternoon doesn't give meaning to the catastrophe, but it lets us glimpse something of value through the meaninglessness.

I first "met" John through a clever article he wrote years ago in *Creative Computing*. I finished the piece, looked at the byline, flipped back to the masthead, and dug out other issues of *Creative* to see who this funny guy was. He was, I saw, someone who saw in computers a tool for the humanities rather than for engineers. He clearly embodied the real meaning of the phrase "creative computing." By the time I met him in person at *MacUser* in 1988, I knew him well.

Sort of. I was surprised to learn how technical he was. In earlier times he had written programming articles, and he had played with hardware to the extent of building a "Hackintosh." He was, I discovered, the founder and sole developer of Acme Dot, one of the first stackware companies.

But the kinds of stacks he did were indicative of his view of computers. They were explorations in new media, artful experiments in animation, and new ways of combining art, design, and writing. He didn't do any calendar or address book stacks, or anything else he had seen elsewhere. He pushed the envelope.

His most recent work was all done under the label of *MacUser*'s Media Lab, which sounds like, and is, R&D. John was pursuing a project on the edge of publishing technology, exploring what publishing could be in the age of computers, considering alternatives to paper and innovative ways to deliver live action, sound, and executable code to subscribers along with the usual words and pictures. He was pushing the envelope.

John was interested, not surprisingly, in the space program, and had lived for a time in Titusville, Florida, near the Kennedy Space Center. He told me once about the auspices under which he arrived there. He had been working for *Creative Computing* magazine when Ziff-Davis folded it, and he needed to find a job. Patch Communications, then the publisher of *Computer Shopper*, was looking for an editor, and was located in Titusville. John was interested. As he was getting off the plane in Florida, he heard some disturbing news on a portable radio someone in the airport was waving around. After he had his rented car and was driving toward Patch, he heard the details of the Challenger disaster on the car radio, but he didn't really need them. All the way to the interview, driving straight toward the Space Center, he could see it.

He went through the interview in a daze. Afterward, he had no recollection of what he had said in the interview.

Everyone knows how the space program was slowed down by the Challenger disaster. Many people drew lessons from the disaster, and it added strength to the argument that we ought to get this planet in order before we go traipsing around the solar system. The most grandiose expression of the fix-earth-first viewpoint is probably the statement "The stars were not made for Man."

I believe that the stars were not made for Man, and recent experience has convinced me that the Earth wasn't, either. But except that we should engineer our buildings and spacecraft carefully, I can't see any significance in either observation.

Pushing back the limits of human achievement, reaching for the stars, that's not something we do. It's what we are.



Michael Swaine
editor-at-large

15 YEARS OF POWER PROGRAMMING!

#161 FEBRUARY 1990 $2.95 (3.95 CANADA)

Dr. Dobb's JOURNAL

SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

WINDOWING SYSTEMS

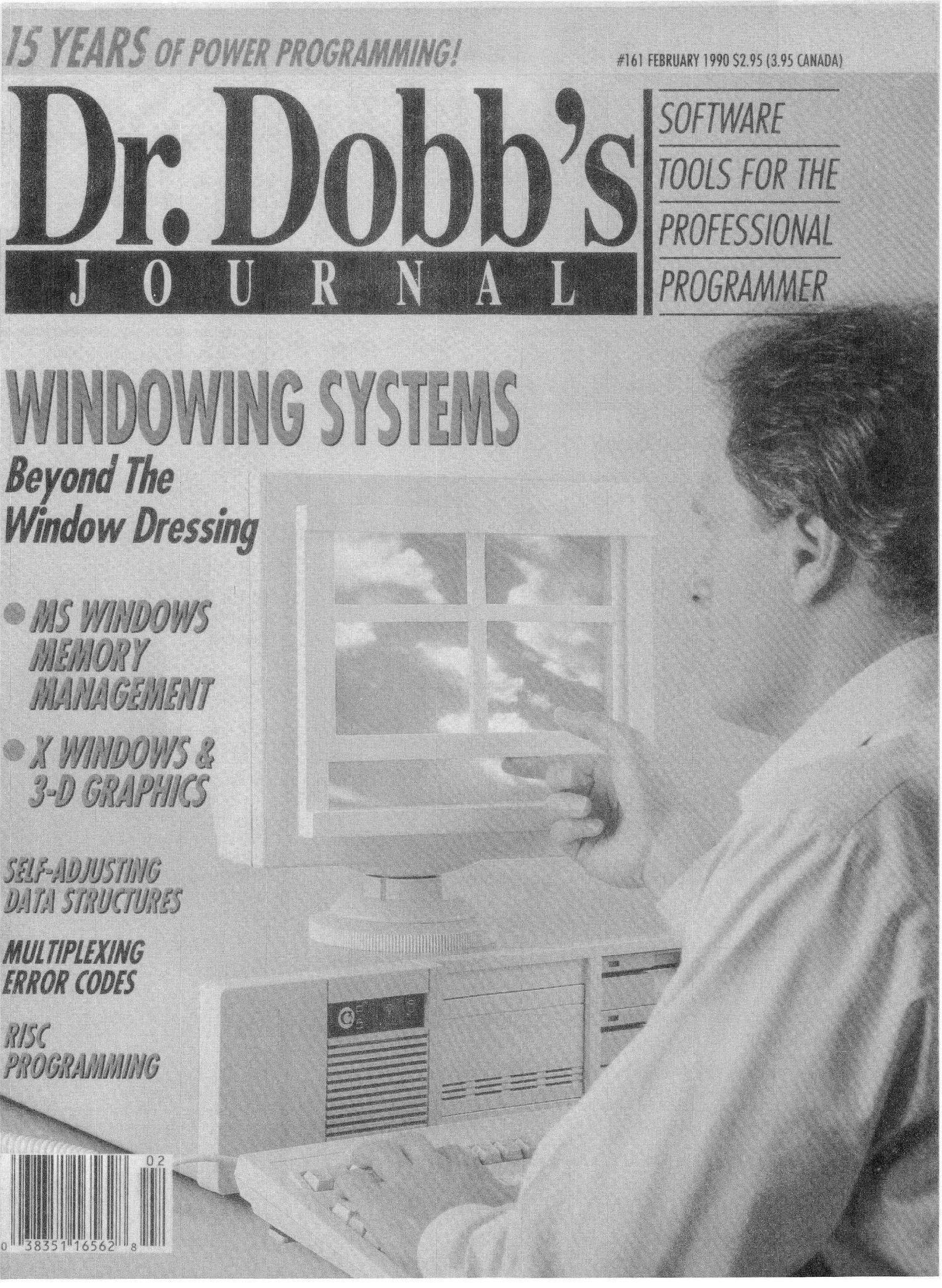
Beyond The Window Dressing

- MS WINDOWS MEMORY MANAGEMENT
- X WINDOWS & 3-D GRAPHICS

SELF-ADJUSTING DATA STRUCTURES

MULTIPLEXING ERROR CODES

RISC PROGRAMMING



02

0 38351 16562 8

CONTENTS

FEBRUARY 1990
VOLUME 15, ISSUE 2

FEATURES

MANAGING MULTIPLE DATA SEGMENTS UNDER MICROSOFT WINDOWS: PART I 16
by Tim Paterson and Steve Flenniken
In the first installment of this two-part article, Tim and Steve present a segment table technique that helps you cope with MS Windows' "memory movement" phenomenon.

THREE-DIMENSIONAL GRAPHICS USING THE X WINDOW SYSTEM 28
by Michael Stroyan
3-D graphics are possible with X Window systems even though most toolkits don't provide much support for creating them. Michael shares his experiences with porting 3-D graphics to X, and provides you with solutions to some thorny problems.

PICK-A-NUMBER INTERFACES 38
by Bob Canup
Sometimes trailing-edge technology provides the right tool for the right job. Bob states his case for when you might choose "pick-a-number" interfaces rather than cutting-edge windowing interfaces.

SELF-ADJUSTING DATA STRUCTURES 44
by Andrew M. Liao
Self-adjusting heuristic algorithms are ideal for lists, binary search trees, and heaps. Andrew explains what they are, and how you can use them.

MULTIPLEXING ERROR CODES 60
by William J. McMahon
It's possible to detect unexpected errors by using function communication techniques such as the one Bill presents here.

PROGRAMMING RISC ENGINES 116
by Neal Margulis
Neal uses Intel's i860 to illustrate how programmers can take advantage of pipelined execution, while Hal Hardenbergh adds his thoughts on RISC vs. CISC.

EXAMINING ROOM

C_TALK/VIEWS 66
by Noel Bergman
In this month's "Examining Room," Noel checks up on C_talk/Views from CNS, focusing on the Browser, Streamliner, Interface Generator, and MVC.

PROGRAMMER'S WORKBENCH

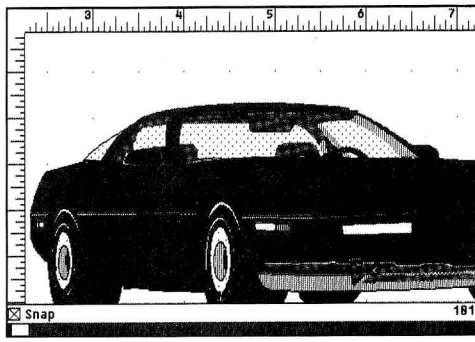
STALKING GENERAL PROTECTION FAULTS: PART II 76
by Andrew Schulman
Andrew continues his hunt for GP faults, this month using 32-bit C compilers and Phar Lap's 386|DOS Extender. He then returns to 16-bit land to see how GP faults can be caught under OS/2.

COLUMNS

PROGRAMMING PARADIGMS 118
by Michael Swaine
This month Mike resumes his examination of Lisp, taking a look at Lisp's representation scheme and the wide range of data structures supported by the Common Lisp standard.

C PROGRAMMING 123
by Al Stevens
Al continues TEXTSRCH, his text retrieval system that provides a concordance-like index into a text data base that uses two general-purpose functions — parsing the command line and binary trees — you might find useful in other projects.

STRUCTURED PROGRAMMING 134
by Jeff Duntemann
Jeff takes time to reflect upon The Quake of '89 before moving on to Arizona and Modula-2.



A special thanks to Bob Ogburn of Halted Specialties Co. (Santa Clara, Calif.) for the equipment used in this issue's photos.

DEPARTMENTS

EDITORIAL 6
by Jonathan Erickson

LETTERS 8
by you

SWAINE'S FLAMES 160
by Michael Swaine

PROGRAMMER'S SERVICES

OF INTEREST 148
compiled *by Janna Custer*

ADVERTISER INDEX 152
where to go for more information on products

PROGRAMMER'S MARKETPLACE 154
classified ads

NEXT ISSUE

March gets *DDJ* back on the assembly line with our lineup of assembly language articles from the perspective of both the 80x86 and 680x0 families. We'll also share some of the tricks that ASM pros use, and we'll take a look at 80386 debugging problems.

Bits and Bytes

Last month's reminiscence about past members of the *DDJ* family brings us to this month's introduction to Ray Valdes, *DDJ*'s new technical editor. Ray joined *DDJ* just before Christmas and, from what I can tell, has pretty much adjusted to coming into an office on a regular basis. For the past few years, you see, Ray's been keeping his own hours as a contract programmer, working on projects ranging from MUMPS to C++ to HyperCard. His specialities, however, are high-end graphics and font technology, topics that go hand-in-glove with his background in graphic design.

If you want to talk with Ray about these or other topics, he can be reached here at the *DDJ* offices, or on MCI Mail and the *DDJ* Listing Service as Rvaldes. His CompuServe ID number will be forthcoming. We're happy that Ray has joined us and pleased with the immediate contribution he's made. You can look forward to seeing and hearing more from him in the future.

Speaking of last month's editorial, about the same time I was rooting through those early issues of *DDJ* and reminding myself of how the magazine began as a forum for Tiny Basic, Microsoft came to town with the release of MS Basic 7.0. Talk about contrasts. The source for Tiny Basic, which consisted of several hundred or so lines of code, was published in a few magazine pages. Basic 7.0, on the other hand, comes with over 2000 pages of documentation and, if you use all of the libraries, help systems, and the like (you don't have to use them all, only what you want) it will claim up to 14 or so Mbytes on your hard disk. Basic 7.0 includes new language features, an integrated environment, new optimization techniques, tools, threaded p-code technology, and on and on.

I'm not at all trying to compare the two, mole hills to mountains being what they are, but instead to point out how far computing and programming have come over a relatively short time. Not only is the software technology there to develop environments like 7.0, but also affordable hardware to support it.

Incidentally, for those of you Basic-kind-of-guys who wondered whatever happened to Borland's Turbo Basic, it has had some new life pumped into it by the folks at Spectra Software (aka PC SIG) and released under the name of PowerBasic. A bunch of new features have been added and if you're interested, give them a call at 408-730-9291.

Back in September, we published *DDJ*'s editorial calendar for the first half of this year. Since we're already two months into that list, it's high time we finish it up.

January	Real-time and Embedded Systems Programming	**July**	Graphics Programming
February	Windowing Systems	**August**	Annual C Issue
March	Assembly Language Programming	**September**	Structured Languages
April	Neural Networks	**October**	Operating Systems
May	Memory Management	**November**	Object-Oriented Programming
June	Hypertext	**December**	Communications and Connectivity

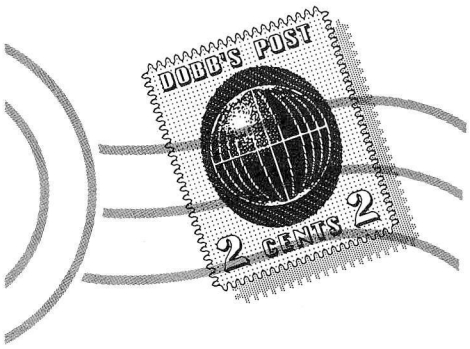
If you have article ideas for any of these topics, give Mike, Ray, or me a call (415-366-3600) or drop us a note (c/o *DDJ*, 501 Galveston Dr., Redwood City, CA 94063, c/o DDJ on MCI Mail, or c/o 76704,50 on CompuServe).

As we expected, Scott Guthery's article "Are the Emperor's New Clothes Object Oriented?"(*DDJ*, December 1989) generated a lot of mail, pro and con. So much, in fact, that we're going to try to pull it all together into an article and continue the debate. Some of the responses have been long (Marshall Giguere's rebuttal was as long, if not longer, than Scott's original), others short and to the point. If there's anything you'd like to add, get it to us as quickly as possible. Or if you'd like to hold off and respond to the responses, that'll be fine too. It should be coming up in a couple of months.

And for those of you who have been wondering whatever happened to the Rhealstone real-time benchmark proposal follow-up we promised last year, Robin Kar has completed the code implementation. We should be running that article within the next two or three months as well.



Jonathan Erickson
editor-in-chief



Standardizing the Standardizing Process

Dear *DDJ*,

Apparently columnist Al Stevens got a little carried away in his November 1989 column when he compared the ANSI standardization process to democracy.

Surely Mr. Stevens knows that a committee is, at best, only a representative democracy. As we in the U.S. are reminded every couple of years, one democratic problem lies in determining who gets to be on the committee. Since I do not remember voting for these people, I wonder whose interests they really represent.

Actually, I commend the C standardization committee for the extent to which they have discussed their issues in print; there has even been some serious discussion of comments from outsiders. In contrast, I was recently involved with comments to the ANSI X3J9 Extended Pascal standardization committee. Since I have read no articles whatsoever on Pascal standardization issues, I wonder how many Pascal programmers realize that such a standard is now virtually complete.

My introduction to Extended Pascal was a note in IEEE *Computer* (Sept. 1988, p. 70) which mentioned that a draft standard was available (for $35). After receiving the draft I became concerned about its content. In fact, I assumed that no group of responsible professionals would consider releasing such a document as an example of their work. But I did submit comments; that was in October 1988.

In June, 1989, I received a package which included copies of all the various comments received, with The Committee's formal response to each point. Apparently I got in on the tail end of the feedback process, since most of the comments were from various other nations' standards organizations, as well as some individuals who were commenting on previous responses. The Committee did decide to make some changes. But my earlier assumption was wrong: For the most part, the draft was allowed to stand.

After an eight-month delay for The Committee to get its responses together, the commentors were given "fifteen working days" to accept or reject their individual responses. Note that the Extended Pascal draft standard is a voluminous, excruciatingly detailed specification, and not everyone can simply drop what they are doing to analyze responses, generate objections, and construct a formal response to The Committee. I did just that, however, and perhaps I was the only one who did, for the response letter that I finally received this September addressed my objections only, and seemed appallingly condescending.

As an engineer myself, I would be the first to admit that some engineers seem to vie with one another to produce the most complex and oblique specification possible. I guess if nobody can understand the spec, then nobody can criticize it; of course, then nobody can help find the bugs in it, either. And I have seen some really awful specifications, with special mention going to IEEE Std 488-1975. But the Extended Pascal draft standard, which purported to describe a structured programming language, was itself so unstructured as to be almost unreadable.

The creation of a specification need not differ from the creation of a program; each must eventually describe a derivative logical system that actually works. A complex system must have a detailed description, of course, but not necessarily a complex organization; that's the whole point of structured programming. In fact, if a system can only be understood in a complex way, how can anyone use it reliably? Simple systems can be reliable, but complex systems always contain errors, unless they can be exhaustively tested by an impartial mechanism, and even then, you never really know.

Some of my comments included the lack of exception handling, the lack of alphanumeric Goto labels, the complete lack of any comment nesting at all, and the lack of multiple *integer* and *real* data types corresponding to word sizes and IEEE reals. I wanted additional C-like looping enhancements such as Break and Continue to avoid the use of unstructured mid-loop Gotos, but The Committee preferred Goto instead. And I was especially concerned over the specification of a single Complex data type with a deliberately undefined implementation. There are two common representations for complex numbers, and each is best suited for different types of computation. The inability to force a particular representation seemed to be a serious programming limitation and a "numerical analysis nightmare." But the particular issues involved are not really the point.

The point in that we have a standardization process functioning largely behind closed doors, without a broad base of general input from the real users of the system under definition. The real users of a Pascal system are individual Pascal programmers. Where are all the comments from these users? When were they contacted? How are they being kept informed? Why were no standardization discussions printed in popular programming magazines? Although I am a member of IEEE, I cannot imagine that this organization is the appropriate sponsor of a programming language standard; a better-suited organization would seem to be the software-oriented ACM.

This experience has taught me a lot about what a standard really means. First of all, there is no reason to think that voting should produce a good standard. Design should be an exercise in consistency and correctness, something not conferred by a winning vote. Indeed, voting for Clarity seems unpopular, possible (cynically) because if a normally intelligent person could simply read the standard, they might not need an expert consultant to help interpret it.

And, when you think about it, democratic voting is itself an adversarial process (indeed, a form of ritualized combat). There may be a substantial minority which is firmly opposed to the outcome, and in scientific or engineering disputes, the minority view may well be correct. Moreover, as in the case of our national political parties, those in the minority have an honorable duty to continue their opposition, if they feel it to be right, against the day when their arguments may prevail. Thus, it is important that not everyone change the way they do things simply because a particular design is voted "standard." And there is certainly no reason to force programming classes to use a particular language version selected by some elitist committee.

At one time there was some discussion in *DDJ* of Borland's Turbo Pascal with respect to the Pascal "standard." I used to wonder why Turbo Pascal did not provide a Standard Pascal mode, but, after studying the Extended Pascal draft standard, now I know: Standard Pascal is a turkey. Although I do have some differences with the Turbo Pascal design, it is a fine language for program expression; it is also clearly described in a structured manner. Per-

(continued on page 12)

(continued from page 8)
haps Turbo Pascal represents the product for the loyal opposition.

Our standards process could be based on the more democratic and obviously more successful Japanese approach of building a widely held consensus, in which case the standards would carry some real weight. But in this Pascal standardization effort, we instead see Ugly American political domination in action. We see back room maneuvering for ANSI institutional support of a particular design, essentially without regard for, or input from, most of the real potential users. This is not democracy, and it may be time for a change.

Terry Ritter, P.E.
Blue Jean Computer Engineering
Austin, Texas

Container Object Fix

Dear *DDJ*,
I really am glad to see the number of OOPS articles that have been in *DDJ* recently. I especially appreciated Anders Hejlsberg's article on container objects ("Container Object Types in Turbo Pascal," Nov. 1989), but have found several errors in the code listings:

In Listing one, function *ListNode.Prev*, change the two references to *Self* to reference the address of *Self (@Self)* as follows:

```
P := @Self;
while P^.Next <> @Self do P := P^.Next;
```

The code will not compile without the above changes, as self is a listnode object type while *P* is a *ListNodePtr* type.

In Listing two, change the with statement in procedure *GetIdent* by adding an address operator to Name as follows:

```
with IdentRefPtr(Idents.Search( @Name,
                                NewIdent))^ do
```

The first parameter to search (the key) should be a pointer.

One useful addition to the list methods is a procedure to concatenate two lists. The circular list type in the contain unit can be efficiently concatenated by swapping the two lists *last.next* pointer.

In a circular list representation, *last.next* points to the first list element. Given the list *{a, b, c}, a.next->b, b.next->c*, and *c.next->a*. Last is a pointer to c. To concatenate list *{d, e, f}*, all we need to do is make *f.next* point to a and *c.next* point to *d*. In other words, the *last^.next* pointers of the two lists need to be swapped. Then the last variables of the lists need to be adjusted.

```
unit Concat;
interface
  Uses Contain; { from DDJ Nov. '89
  type
  CatListPtr = ^CatList; CatList =
                                object(List)
    procedure cat(L: CatListPtr); end;
  implementation
  Procedure CatList.Cat(L: CatListPtr);
  { Concatenate list L to the list invoking the method - L is appended to the
END of the list being acted on. }

Var P: ListNodePtr; { temporary } Begin
  { swap the two pointers }
  P := Last^.Next;
  Last^.Next := L^.Last^.Next;
  L^.Last^.Next := P;

  { adjust pointers }
  Last := L^.Last;
  L^.Last := NIL; End; END.
```

Eric Friedman
Glenview, Illinois

Delving into Drive Paths

Dear *DDJ*,
I have been able to make a great deal of use of the information in Mr. James' article, "Undocumented DOS," June, 1989. In the process, I have determined a more precise definition of the Drive Path Table that he presented. Perhaps some of your readers will find this information as useful as I have.

The Drive Path Table is as follows. (Note that some of the following information is in direct conflict with the information presented by Mr. James.)

The number of entries in the Drive Path Table is set by the LASTDRIVE=xx command in your CONFIG.SYS file. To access the Drive Path Table for a particular drive, start with the drive number (0 =A, 1=B, . . .) and make sure it is less than the LASTDRIVE=xx value; if it is not less, the drive number is invalid. (See Mr. James' article, Table 2, byte 21H.) If the drive number is valid, multiply it by 51H and add that value to the Drive Path Table pointer (same article, same table, offset 16H). The resulting 32-bit value points at the Drive Path Table entry for that drive.

For normal drives, the pathname field contains the current default pathname (e.g. C:\MY_PATH), the flags field has the value 4000H, and the root-length field has the value 2.

For unused drives, the flags field has the value 0. All remaining fields are set to all 0s or all 1s, except the pathname field, which contains the drive letter, a colon, and a '\'.

For drives created with the SUBST command, the pathname field contains the target path, the flags field has the value 5000H, and the root-length field is set to the length of the target path. For example, after executing the command SUBST E: C:\BIN the pathname field for drive E would be "C:\BIN", and the root-length field would have the value 6.

Now, if you change directories on drive E (e.g. CD E:NEXT) the pathname field is updated (C:\BIN\NEXT) but the root-length field remains unchanged at 6.

If you use the JOIN command on a drive, the pathname field contains the path to which the drive is joined, the flags field contains the value 6000H, and the root-length field is set to 2. For example, after the command JOIN A: C:\DRIVEA the pathname field for drive A would be "C:\DRIVEA". Because drive A is no longer a valid drive, you cannot change the current default directory of this drive. If you CD to C:\DRIVEA\ANYPATH, that pathname is stored in the Drive Path Table entry for drive C, not A.

For network drives, the pathname field contains the network name, followed by the network path; the network name is preceded by two backslashes. The flags field contains the value 0C000H, and the root-length field is set to the length of the combined network name and path strings. For example, after the command NET USE F: \\MYNET\MYPATH the pathname field for drive F would contain "\\ MYNET\MYPATH", and the root-length field would contain the value 14.

Now, if you change directories on drive F (e.g. CD F:NEXT) the pathname

(continued on page 14)

Offset	Size	Description
0	byte [67]	pathname: an ASCIIZ string.
43H	word	flags: 8000H a network drive. 4000H this entry is valid. 2000H a JOIN drive. 1000H a SUBST drive.
45H	dword	pointer to the drive parameter block.
49H	word	block/track/sector information.
4BH	dword	unused (set to 0FFFFFFFFH)
4FH	word	root length.

(continued from page 12)
field is updated (\\MYNET\MYPATH\NEXT) but the root-length field remains unchanged at 14.

For network drives, the last few fields of the Drive Path Table are redefined slightly. Beginning at offset 45H, the Drive Path Table becomes:

45H	dword	unused (set to 0)
49H	dword	pointer to a network drive table
4DH	word	saved parameter
4FH	word	root length

The pointer at offset 49H points into a linked list of network devices (drives and printers).

The saved-parameter field stores a user-defined value. This value is selected when the drive is created. (See INT 21H function 5F03H.) I suppose it would be used by someone writing a network driver, who needed to keep track of additional information about the drive.

When accessing the Drive Path Table, you should be aware of the ASSIGN command. If the ASSIGN command has been invoked, it will automatically translate drive letters that are passed through the conventional INT 21H function calls. But, it will NOT translate these drive letters as you access the Drive Path Table directly.

In DOS 3.xx, you can determine whether ASSIGN is installed (ASSIGN is a TSR) like this:

```
MOV     AX,0600H
INT     2FH
OR      AL,AL
JZ      NOTINSTALLED
88
```

If ASSIGN is installed, you can get a pointer to its translation table like this:

```
MOV     AH,19H
INT     21H
PUSH    AX; save the current
        default drive
MOV     AX,0601H
INT     2FH
POP     DX; restore the drive
MOV     AH,0EH
INT     21H
MOV     AL,ES:[0103H+drivenumber1]
        ; drivenumber: 1=A, 2=B, . . .
```

The translation table is always 26 bytes. By default, each drive is assigned to itself (e.g. entry 1=1, entry 2=2, . . .). After an assignment such as ASSIGN A= C, entry 1 gets the value 3, and then all attempts to access drive A are routed to drive C.

Thus, whenever you access the Drive Path Table, you should index the drive number into the ASSIGN translation table, if it exists. And if you use the pathname field, you should reverse-assign the drive letter therein.

Reverse-assigning a drive letter means scanning the ASSIGN translate table to find a drive number that, when translated, will result in the drive letter that appears in the Drive Path Table pathname field. And be prepared that you may not be able to find such a drive number.

For DOS 2.xx, I do not know of a reliable way of determining whether the ASSIGN command is installed. The ASSIGN command tries to detect itself, but the method it uses is not the elegant INT 2FH interface, and it fails under certain circumstances.

To detect itself, DOS 2.xx ASSIGN looks at the first few bytes pointed at by the INT 21H interrupt vector. If those bytes match the first few bytes of ASSIGN's INT 21H handler (80, FC, 0C, 76, 2D, 80, FC, 50), ASSIGN decides that it is already installed. Rather than installing a second copy of itself, ASSIGN updates the installed translation table, and exits quietly. (To access the installed translation table, ASSIGN uses the segment value from interrupt vector 21H and an offset value of 0103H.)

The problem is that, after installing ASSIGN, if you install any TSR that also has an INT 21H handler, this detection scheme fails. In fact, if you run the ASSIGN command again under these circumstances, it will install itself a second time. This will likely cause strange results, because the drives may be translated multiple times.

If, when working with drives, your program uses exclusively one method or the other (that is, either conventional interfaces or undocumented interfaces, but not both) then you probably need not worry about the ASSIGN command.

If you must use both interfaces, and you want your program to work correctly in DOS v2.xx, you should warn the program's users about the restrictions regarding the DOS 2.xx ASSIGN command.

In summary, the Drive Path Table allows access to information that would not otherwise be available to normal programs (i.e. the JOIN and SUBST information.)

Michael Cook
Mitel Semiconductor
Kanata, Ontario

Graphics For the Rest of Us

Dear *DDJ*,

I found the interview with David Parker in Michael Swaine's column entitled "Parker's Perceptions" in the October 1989 *DDJ* especially interesting since I'm a very satisfied user of his "AcroSpin" graphics software. I'd known nothing about the person behind it!

Swaine says "it seems to have a good product," and I can testify that it does indeed. My colleagues and I need to do 3-D graphics in a variety of languages, on a number of different IBM PC compatible machines. We also cannot afford fancy hardware; my own machine doesn't even have a hard drive.

AcroSpin is the only product I've seen that does the things I need on the hardware I've got. On all our PC's at this institution, I've yet to encounter one that has some kind of graphics and cannot run AcroSpin. It's clean, simple, and does exactly what the manual says it'll do.

Matthew D. Healy
Zoology graduate student
Duke University, North Carolina

FPCA '89 Proceedings

Dear *DDJ*,

I very much enjoyed Ronald Fischer's report on functional programming and FPCA '89. I must take issue with Mr. Fischer on one point. He states that "Scheme is a deviation of Lisp that doesn't offer lazy evaluation." Scheme certainly does offer delayed (lazy) evaluation, via the DELAY and FORCE primitives. See, for instance, Chapter 3 of Abelson, Sussman, and Sussman's *Structure and Interpretation of Computer Programs* for a superb introduction to streams and delayed evaluation in Scheme.

I was disappointed that the article did not mention how to obtain the conference proceedings. Could you provide this information?

David Cabana
Tampa, Florida

DDJ: Sure, David. If you are a member of the ACM (Association for Computing Machinery), you can obtain a copy through them for $22. Just call 800-342-6626. If you are not a member, you'll have to pay a little extra and buy the book from Addison-Wesley for $35.50. Their number in Reading, Mass., is 617-944-3700.

We welcome your comments (and suggestions). Mail your letters (include disk if your letter is lengthy or contains code) to DDJ, 510 Galveston Dr., Redwood City, CA 94063, or send them electronically to CompuServe 76704,50 or via MCI Mail, c/o DDJ. *Please include your name, city, and state. We reserve the right to edit letters.*

Part I

Managing Multiple Data Segments Under *Microsoft Windows*

Segment tables can help you manage multiple data segments under MS Windows

Tim Paterson and Steve Flenniken

If you've ever done any serious program development work for Microsoft Windows, you're probably aware of the freewheeling attitude Windows takes with memory management. In short, nothing stays put!

In this two-part article, we'll present a technique that helps you cope with this "memory movement" phenomenon by using a little-known Windows feature, the segment table. In this month's installment, we'll begin by defining the problem and identifying the solution, including a short library of macros and functions that will make the technique easy to use. Next month, we'll have a working sample application that demonstrates both the use of the segment table and our library, and provides a window (figuratively and literally) into the operation of the segment table while Windows is running.

The memory movement activity described above is particularly obvious in a Windows debugging session under Symdeb. Especially when Symdeb has been asked for full memory movement reporting (*/W3*), the memory movement/allocation/discard messages just spew out on the debugging screen. The purpose of all this activity is to utilize global memory as efficiently as possible. Each item being manipulated is a segment that was defined by the programmer either at compile and link time, or at run time. The offset of a particular item within one of these segments is not affected by movement of the segment. (Offsets within the local heap in the default data segment can change, but this is independent of global memory management.)

While these segments are being moved around, Windows keeps the values in *DS* and *SS* updated. *SS* is always the default data segment (*DGROUP*), and *DS* usually is, too. But *DS* could be any segment known to Windows, and if Windows moves that segment, *DS* will reflect that change. However, some segments (particularly code segments) are discardable; if *DS* is set to a segment that is later discarded, there will be no updating and no warning. Note that *ES* is never updated for segment movement.

Because *SS* and *DS* are kept current by Windows, near pointers into the stack or to static data are not a problem. Far pointers, on the other hand, seem to become invalid at every opportunity available to Windows. Fortunately, Windows does not interrupt your code and move things; any block of instructions that doesn't give control to Windows does not have to worry about memory movement for its duration. Windows gets control from a program (and possibly moves memory) in three ways. First, and most obvious, the program gives Windows control whenever it calls a Windows function. Second, almost any far call the program makes could also give control to Windows. Third, almost any far return might do it as well. These second and third cases occur because code segments are normally marked as discardable, and Windows has positioned itself to grab control and load any code segment that's not in memory when it gets called or returned to. This cannot happen when calling/returning into a fixed segment, or when calling/returning into the currently executing segment, since in each case the target segment is already present.

At face value, it would appear that you could never pass a far pointer as a parameter nor store it in a pointer variable,

Tim is the original author of MS-DOS, Versions 1.x, which he wrote in 1980–82 while employed by Seattle Computer Products and Microsoft. He was also the founder of Falcon Technology, which was eventually sold to Phoenix Technologies, the ROM BIOS maker. Steve formerly worked at Seattle Computer Products, Rosesoft (makers of ProKey), and is now with Microrim, working with OS/2 and Presentation Manager. They can be reached c/o DDJ.

because it wouldn't be valid by the time it got used. (One exception is far pointers to data in *DGROUP*, which will not be invalidated when passed to Windows functions.) The standard solution for seasoned Windows programmers is to lock the segments of the data down, so that they won't move. Locked segments, however, are blockages that choke up Windows' memory management and hurt performance. A well-behaved Windows application will lock no more than a few segments at a time, and then only for a short period. Seasoned Windows programmers know about all this stuff because they've all read Chapter 8 of Charles Petzold's *Programming Windows*.

There is another approach that is useful only for passing parameters between assembly language routines because it violates the C calling convention. A single far pointer can be passed by using *DS* to carry the segment, passing the offset in another register or on the stack. As mentioned already, Windows will keep *DS* updated while it moves memory around, as long as it doesn't point to a segment that gets discarded. There is, however, a restriction on the value in *DS* at the time of a far call from (not to) a discardable code segment. *DS* must contain a segment that is subject to Windows' memory management, that is, one that has a Windows handle associated with it. Segments outside of Windows' memory management would include low memory (such as *40H*, where BIOS data are stored) or expanded memory (say, at segment *E000H*) that is managed directly by the application. This restriction is needed should the caller's code segment be discarded. When this happens, the return address on the stack is patched so that it will transfer to code in Windows that reloads the caller. As part of this patching, the saved value of *DS* in the stack gets replaced by its handle. A handle does not require all 16 bits, and the extra bits made available by this replacement are used to store some of the segment reload and return information.

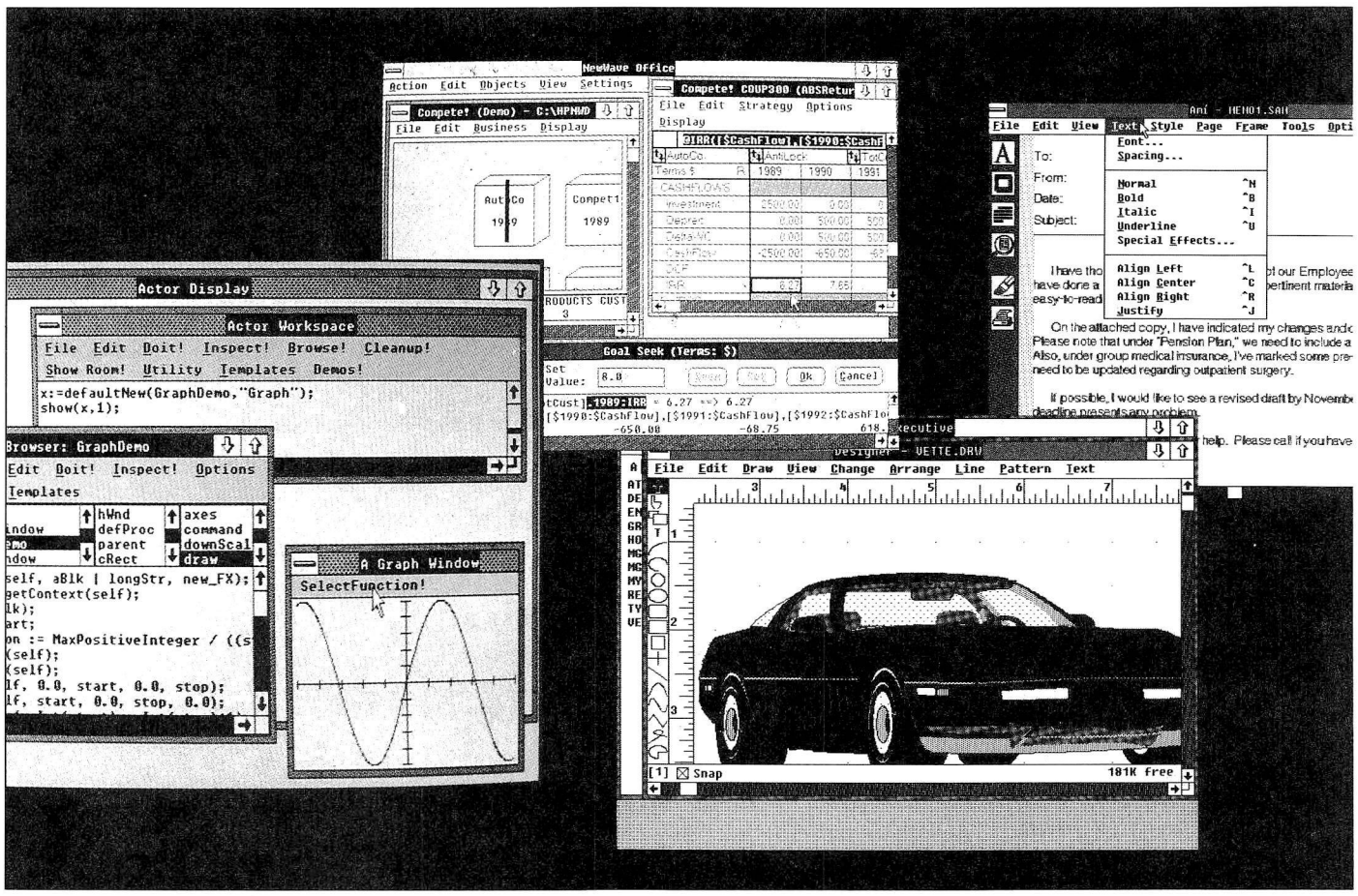
Big Applications

Let's consider one of the big applications — spreadsheets, word processors, data base managers, and integrated programming environments — that might (someday) be run under Windows. Taking a spreadsheet program as an example, we can speculate on how its data segments might be organized.

First, each spreadsheet needs at least one data segment for its cell values and formulas. Considering the huge size that modern spreadsheet programs can theoretically attain (16,384 rows × 256 columns > 4 million cells), many segments of 64K could be required. After all, each cell will take many bytes to represent a formula or constant.

But besides a segment for cell information, there are other kinds of data tied to each spreadsheet. For example, names can be defined to refer to cells within the sheet, and you need to keep track of which segments constitute the cell data. This stuff might all go into a "control" data segment for each spreadsheet.

The spreadsheet program would allow several sheets to be loaded at once, each in their own window. Each spreadsheet loaded will require at least two data segments. The program-wide data, such as the list of information (e.g., name and path) about each loaded sheet, should probably go into its own segment, rather than into *DGROUP*. The rule we're following is that data that grow at the whim of the user — like loading more sheets — should not be crammed into *DGROUP*, as it already contains plenty of stuff. It's embarrassing to have a program report "out of memory" because *DGROUP* is full, when there is really 200K of global



memory left.

The point is, a big application will need a lot of data segments. And the data are mostly structures which will get passed around by reference. In other words, far pointers will be necessary everywhere. And Windows will be happy to move memory around and invalidate these pointers before they ever get used. Or the segments can be locked, and the application will turn into a memory pig.

The Segment Table

A little-known alternative to this dilemma has been built into Windows specifically to help with these problems of multiple data segments. Because a description does not appear in the Windows Software Development Kit, we will create our own terminology in order to explain it. With this method, segments are never locked, so Windows has the freedom to manage memory for optimum efficiency. Segment references are passed around and stored indirectly, but can be dereferenced into a hard segment number instantly, without calling a Windows function.

The heart of the mechanism is the segment table. This is an array of words in *DGROUP* that the application program registers with Windows by calling the Windows function *DefineHandleTable(pSegTable)*. The words in the segment table are the segment numbers of all the data segments used by the application. Windows will keep these segment numbers updated and valid as memory is moved around.

To use this technique, the application must first set aside a static array of words in *DGROUP*. Source code in Microsoft C might look like that in Example 1.

The first two words in the segment table have special meaning, while the rest can all contain segment numbers. The length of the array is set to *MAXPSEGS + 2* to account for this. The array declaration assumes that the Small or Medium model is being used, which is typical of Windows applications. If the Compact or Large model is used, *NEAR* should be added to assure allocation in *DGROUP*.

The first word of the segment table, *cwSegs*, contains the number of segment entries in the table. This is the number of words Windows scans, looking for the segment it just moved. If it finds it, Windows will replace the old value in the table with the new location of the segment. Windows will continue to scan the table in case the segment appears in it more than once.

In the sample code in Example 1, an initializer could have been used in the declaration of *SegmentTable* to set *SegmentTable[0]* (*cwSegs*) to *MAXPSEGS (30)*. However, it's more efficient to make *cwSegs* only as large as needed to include all active entries in the table. For example, if only the first five entries in the segment table actually have segments it them, there is no point in having Windows scan 30 entries every time it moves memory. The initial value for *cwSegs* could be zero (as shown here), or it could be the smallest number of segments the application could need.

```
/* WORD defined as unsigned short in windows.h */
#define MAXPSEGS 30  /* max # of table segments */
WORD SegmentTable[MAXPSEGS+2];
#define cwSegs  SegmentTable[0]
#define cwClear SegmentTable[1]
```

Example 1: *Sample source code for setting aside the segment table*

```
#define segDgroup       SegmentTable[2]
segDgroup = HIWORD((void FAR *)&SegDgroup);
++cwSegs;  /* was zero, now 1 */
```

Example 2: *Sample code for putting the group into the table*

```
if ( hMem = GlobalAlloc(flags, dwBytes) )
   SegmentTable[i] = HIWORD( GlobalHandle(hMem) );
else
   /* out of memory */
```

Example 3: GlobalHandle( ) *converts a handle to a segment*

The application program's initialization code will register the table with Windows. For example:

```
void FAR PASCAL DefineHandleTable(WORD NEAR *);
                                    /* prototype */
. . .
DefineHandleTable(SegmentTable);
```

When this call is made, Windows will zero out *cwClear* and the following *cwSegs* entries in the table. The application can then start putting segment numbers in the table, which Windows will keep updated. The first segment the application will want in the table is *DGROUP* itself. Although the correct value of *DGROUP* is always maintained in segment register *SS* (and usually *DS*), it can be helpful to have it in the table as well, for reasons we'll explain later. This would require code like that in Example 2.

Finding these empty entries is simple and involves no searching. Keeping cwSegs *as small as possible is important for good performance, because Windows scans the whole table for every memory movement*

All data segments, besides the default data segment *DGROUP*, must be allocated with the Windows function *GlobalAlloc( )*. *GlobalAlloc( )* (Example 3) returns a handle, not a segment, and traditionally a call to *GlobalLock( )* is used to get the segment and lock it. We don't want it locked, however, so a call to *GlobalHandle( )* should be used instead. *GlobalHandle( )* is sort of a strange beast because its argument can be either a handle or a segment — it will figure out which. (This is actually pretty easy, because handles are always even numbers, and segments are always odd.) Given a handle or a segment, *GlobalHandle( )* returns both: The low word of the return value is the handle, and the high word is the segment.

Using Fixed Table Entries

If we were building a big application such as a spreadsheet, we would probably always need some data segments in addition to *DGROUP*. Let's suppose we always have two additional data segments. One of them will store the full path name of each spreadsheet file that is currently loaded. The other segment will have some kind of "descriptor" for each loaded sheet, including the offset of its name in the first segment. Because these segments are always there, we'll assign them to the next two locations in *SegmentTable* after *DGROUP*.

```
#define segPathNames        SegmentTable[3]
#define segSheetDescr       SegmentTable[4]
```

(continued from page 18)

Initialization procedures should call the Windows functions *GlobalAlloc( )* and *GlobalHandle( )* to assign values to these table entries. Note that the value in *cwSegs* (*SegmentTable[0]*) must be maintained at no less than the number of entries in the table being used. At the latest, *cwSegs* can be updated immediately after the segment number is stored in the table — before any function call or return is made. For example, after the first segment number of these two has been stored, *cwSegs* must be no less than 2; thus the first two entries in *SegmentTable* (*segDgroup* and *segPath Names*) will be kept updated as memory is moved during the function calls that allocate *segSheetDescr*. After *segSheet Descr* has been set, *cwSegs* must be no less than 3.

I've been saying "no less than" because it's OK if *cwSegs* is bigger than what's needed. In fact, the simplest approach for this example would be to initialize it to 3 in the declaration of *SegmentTable*. The effect of this is that Windows will scan three entries whenever it moves memory around to see if any of them are the segment it just moved. Because external data in C is initialized to zero, it will never find a match on the segment number, and nothing will happen. But it doesn't hurt even if garbage is present in the scanned part of *SegmentTable* — if a match is found, Windows will update that location with the new segment number. New garbage in place of the old — no problem, so long as *cwSegs* is still within the total size set aside for *SegmentTable*.

Some macros can be a big help in putting the segment table to use. The segment from the segment table must be combined with an offset into a long integer that is typecast to a far pointer as shown in Example 4. Another typecast is needed when the pointer is dereferenced to specify the type being pointed to. If there are many references to a particular type, a macro for that type can reduce some of the clutter as shown in Example 5.

This idea can be extended one step further. Some of the entries in the segment table have a dedicated meaning, which could be written into a macro. Suppose that there was a separate structure in the segment *segSheetDescr* for each spreadsheet that's been loaded. Because these structures can only exist in that one segment, the access macro can have that segment built in, as in Example 6.

In actual practice, you might want to copy a structure such as this into a fixed location in *DGROUP* to improve both the code size and the speed of repeated accesses. This particularly makes sense when there is a single "current" structure that is used heavily, which certainly applies to the spreadsheet case. Those structures that aren't current can still be accessed with the additional overhead.

```
/* HIWORD defined in windows.h */
#define FARPTR(off,seg) ((void FAR*)MAKELONG(off,seg))
*(WORD FAR *) FARPTR(0,segSheetDescr) = 0;
```

Example 4: *Macro for creating a file pointer*

```
#define FARWORD(off,seg) (*(WORD FAR*)FARPTR(off,seg))
FARWORD(0,segSheetDescr) = 0;     /* no sheets loaded */
```

Example 5: *Macro for referencing a particular type*

```
#define FARDESCR(off) (*(DESCR FAR*)FARPTR
                              (off,segSheetDescr))
typedef struct  /* descriptor for a spreadsheet */
{
          unsigned short              SheetId;
          char                        *SheetName;
          ...
} DESCR;

FARDESCR(pCurSheet).SheetId = NextSheetNum++;
pCurSheetName = FARDESCR(pCurSheet).SheetName;
```

Example 6: *Building in a segment*

```
if ( hMem = GlobalAlloc(flags, dwBytes) )
{
   pSeg = &SegmentTable[++cwSegs + 1];
   *pSeg = HIWORD( GlobalHandle(hMem) );
}
else
   /* out of memory */
```

Example 7: *Allocating a segment*

```
typedef unsigned long IFP;
#define MAKEIFP(off,pseg) ((IFP)MAKELONG(off,pseg))
#define IFP2SEG(ifp) ( *(WORD NEAR *)HIWORD(ifp))
#define IFP2PTR(ifp) FARPTR(LOWORD(ifp),IFP2SEG(ifp))
```

Example 8: *Defining a new data type*

```
IFP memcpyifp(
  MAKEIFP(&CurSheetBuf,&segDgroup), /* destination */
  MAKEIFP(pCurSheet,&segSheetDescr), /* source */
  sizeof(DESCR) /* bytes to copy */
 );
```

Example 9: *A call to a general-purpose block copy routine*

Variable *pSegs*

So far, we have only talked about a few dedicated entries in the segment table. But a big application will need to grab and release new data segments on the fly. For example, earlier we proposed that each spreadsheet could need two segments, and any number of spreadsheets could be loaded at once. We need a method to allocate new entries in the segment table as spreadsheets are loaded.

There's no trick to this part. Windows doesn't care which entries in the segment table we use. One simple-minded approach would be to use *cwSegs* to keep track of the next free entry. For example, if *cwSegs* is 5, then *SegmentTable[2]* through *SegmentTable[6]* are in use. *SegmentTable[7]* would be the next free entry, and *cwSegs* would need to be incremented when it's allocated, as shown in Example 7.

The final result of the allocation is *pSeg*, which is a pointer to a segment. This replaces the Windows handle as the way to remember a segment. If you wanted to set the first word in a newly allocated segment to zero, you could use

```
FARWORD(0,*pSeg) = 0;
```

Now that we're using *pSegs*, we'll need a new data type to represent a *pSeg* combined with an offset. Let's call it an "Indirect Far Pointer," or IFP. IFP is a 32-bit quantity, with an offset in the low word and a near pointer to a segment in the high word, as in Example 8.

The IFP is the quantity that will be passed as a parameter to other functions in the program. Note, however, that the IFP cannot be passed to Windows nor to any standard C libraries. Example 9 shows an example of a call to a general-purpose block copy routine that uses IFPs to specify the source and destination. Except for using IFPs, this function has the same inputs, outputs, and purpose as the standard Microsoft C library routine *memcpy( )*. The first argument is the destination, in this case the static variable *CurSheetBuf* of type *DESCR*, which holds the whole descriptor for the "current" spreadsheet. Now you can see why an entry in the

(continued from page 20)
segment table is reserved for *DGROUP*, even though its current value can always be found in segment register *SS*. Because the copy routine is general purpose, it must always

```
IFP memcpyifp(IFP DestIfp, IFP SourceIfp, unsigned cb)
{
    unsigned i;
    for (i=0; i<cb; i++)
              ((char FAR *)IFP2PTR(DestIfp))[i] =
                ((char FAR *)IFP2PTR(SourceIfp))[i];
    return(DestIfp);
}
```

Example 10: *One way to code* memcpyifp( )

```
          char FAR *Dest;
          char FAR *Source;

          Dest = IFP2PTR(DestIfp);
          Source = IFP2PTR(SourceIfp);
          for (i=0; i<cb; i++)
                        Dest[i] = Source[i];
```

Example 11: *Storing pointers in local variables*

```
IFP memcpyifp(IFP DestIfp, IFP SourceIfp, unsigned cb)
{
   movedata( IFP2SEG(SourceIfp), /* source seg */
   LOWORD(SourceIfp),     /* source offset */
   IFP2SEG(DestIfp),      /* destination seg */
   LOWORD(DestIfp),       /* destination offset */
   cb                     /* count of bytes */
  );
 return(DestIfp);
}
```

Example 12: *Writing* memcpyifp( ) *using a standard library function*

be passed IFPs, which are *offset/pSeg* pairs. In order to pass a *DGROUP* variable, such as *CurSheetBuf*, we must be able to pass a pointer to a place where *DGROUP* is stored.

In this example, the source of the copy is the descriptor that is being made current. *MAKEIFP* builds the reference for the source by using the offset held in the static variable *pCurSheet*, and the dedicated *pSeg segSheetDescr*. Note how the "address of" operator "&" applied to *segDgroup* or *segSheetDescr* returns a *pSeg*; leaving it off would fetch the segment value itself from *SegmentTable*.

One way to code *memcpyifp( )* is shown in Example 10. Unfortunately, the Microsoft C compiler is not as good at eliminating loop invariants as we might like. Even at maximum optimization (*/Ox*), the generated code completely rebuilds both far pointers — and fetches the segments from the segment table — each time through the loop. A slight variation, then, would be to bring the building of the far pointers out of the loop and store them in a couple of local variables, such as those shown in Example 11.

The code generated for this sequence is pretty good, and ends with the block move instruction *REP MOVSW* to actually perform the copying. It is critical, however, to remember the limitations of applying this technique to other cases. Once an IFP has been dereferenced, there is no longer any protection from global heap movement. Almost any far call can be assumed to invalidate far pointer variables. An exception to this rule is when far calls are made to functions in the same segment; because the segment is already loaded, no memory movement will occur. This allows us to write "helper" functions to access standard C library routines. The helper must reside in the same segment as the run time, and converts the IFPs passed to it into far pointers for the library

(continued from page 22)
function. As shown in Example 12, *memcpyifp( )* could be written using the standard Microsoft C library function *movedata( )*, which takes explicit segment and offset arguments.

For the best performance and smallest size, key functions can be coded in assembly language. Listing One, page 89, shows an assembly-language version of *memcpyifp( )*, demonstrating the dereferencing of IFPs.

The *segtable* Library

Listing Two, page 89, shows the *segtable* library, a collection of five functions that manage memory through a segment table. The header file segtable.h (Listing Three, page 90) includes all the function prototypes, type definitions, and macro definitions needed to use the library. The first few lines of segtable.h are constants that should be tailored to the application: MAXPSEGS is the maximum number of segments that the segment table can hold. You want this large enough so you never run out, but you don't want to eat up too much of *DGROUP* with the table. Microsoft Excel, for example, apparently has room for more than 2000 segments. *MINPSEGS* is the initial value for *cwSegs*. It must be equal to the number of fixed table entries — at least 1, for *segDgroup*.

Immediately following these constants are the definitions of the fixed table entries. *segDgroup* is always there as *SegmentTable[2]*, and any additional entries must follow with consecutive indices.

The header file also defines a pair of validation macros for debugging purposes. *VALIDPSEG(pseg)* verifies that its *pSeg* argument points into the segment table and is word-aligned (even). *VALID_VARIABLE_PSEG(pseg)* adds the additional requirement that the *pSeg* point above the fixed table entries, into the variable part of the table. The library routines make these checks whenever possible if the *DEBUG* switch is turned on. Also, whenever a segment table entry contains a valid segment number, there are checks to verify that the entry is odd (as are all Windows segment numbers). If any of these debugging tests fails, the library will call the function *SegmentError( )*.

The five library functions are described in the following section.

void FAR PASCAL SegmentInit(void) — This function registers the segment table with Windows and initializes the *segDgroup* entry. All other fixed table entries will have a zero value, which is appropriate for allocating with *SegmentRealloc( )*.

PSEG FAR PASCAL SegmentAlloc(DWORD size) — First, this function must find an unused entry in the segment table. It does not use the simple method described earlier; instead, it keeps a free list that links together all entries that were once used but have been freed. A static variable points to the head of the list. The segment table is located at an even address, so all the links in the free list will also be even. This means that when Windows scans the table for a segment it is moving, it will never match with a link because the segments are always odd. If the free list is empty, then *cwSegs* is increased and the next entry at the end of the table is taken.

This approach to finding unused entries ensures that *cwSegs* is never increased if there is already a free slot within the in-use range. Finding these empty entries is very simple and involves no searching. Keeping *cwSegs* as small as

Dr. Dobb's JOURNAL
SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

PUBLISHER *Peter Hutchinson*

EDITORIAL
EDITOR-IN-CHIEF *Jonathan Erickson*
MANAGING EDITOR *Monica E. Berg*
TECHNICAL EDITORS *Michael Floyd, Ray Valdes*
EDITORIAL ASSISTANT *Janna Custer*
CONTRIBUTING EDITORS *Al Stevens, Jeff Duntemann, Martin Tracy, David Betz, Tom Genereaux, Andrew Schulman*
COPY EDITORS *Rosalie Cooke, Pamela Dillehay, Nan Fornal*
EDITOR-AT-LARGE *Michael Swaine*

ART/PRODUCTION
ART/PRODUCTION DIRECTOR *Larry L. Clay*
ART DIRECTOR *Michael Hollister*
PRODUCTION SUPERVISOR *Amy Shulman Lesovoy*
TECHNICAL ILLUSTRATOR *Linda Ann Clark*
TYPOGRAPHERS *Teresa Raines, Margaret Anderson, Charlene Carpentier*
COVER PHOTOGRAPHER *Michael Carr*

CIRCULATION
DIRECTOR OF CIRCULATION *Maureen Kaminski*
CIRCULATION MANAGER *Randy Robertson*
CIRCULATION PLANNING MANAGER *Manny Sawit*
DIRECT MARKETING MANAGER *Andrea Weingart*
NEWSSTAND MANAGER *Sarah Forsman*
DIRECT MARKETING COORDINATOR *Francesca Davies*
PROMOTION COORDINATOR *Pam Moore*
FULFILLMENT COORDINATOR *Anne Jean*

ADMINISTRATION
VICE PRESIDENT OF FINANCE *Kate Deschamps*
CONTROLLER *Mary Collopy*
CREDIT MANAGER *Betty Arsene*
ACCOUNTING SUPERVISOR *Renate Kernke*
ACCOUNTS RECEIVABLE *Wendy Ho*
ACCOUNTS PAYABLE *LuAnn Rocklewitz*

MARKETING/ADVERTISING
DIRECTOR OF SALES AND MARKETING *Karla Spormann*
ADVERTISING COORDINATOR *Laura Stack Pullen*
MARKETING ASSISTANT *Sara Noah Ruddy*
ACCOUNT MANAGERS *see page 152*

M&T PUBLISHING INC.
CHAIRMAN OF THE BOARD *Otmar Weber*
DIRECTOR *C. F. von Quadt*
PRESIDENT *Laird Foshay*
VICE PRESIDENT OF PUBLISHING *William P. Howard*
VICE PRESIDENT/GROUP PUBLISHER *Randall L. Stickrod*

DR. DOBB'S JOURNAL (USPS 307690) is published monthly, except semimonthly in December, by M&T Publishing, Inc., 501 Galveston Dr., Redwood City, CA 94063; 415-366-3600. Second-class postage paid at Redwood City and at additional entry points.

ARTICLE SUBMISSIONS: Send manuscripts and disk (with article and listings) to the editorial assistant 415-366-3600.

***DDJ* ON COMPUSERVE:** Type GO DDJ.

***DDJ* LISTING SERVICE:** 603-882-1599. Supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Type *listings* (use lowercase) at the login prompt.

SUBSCRIPTION: $29.97 for 1 year; $56.97 for 2 years. Foreign orders must be prepaid, including the additional postage (air or surface) in U.S. funds drawn on a U.S. bank. Add $13 for surface mail to all addresses out of the U.S.; add $33 for airmail to Canada and Mexico; or $32 for airlift to all other countries.

POSTMASTER: Send address changes to *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. **ISSN 1044-789X**

CUSTOMER SERVICE: For subscription orders and changes of address call toll-free 800-456-1215 or write *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. For subscription questions outside the U.S. call 1-303-447-9330. For book/software orders call 800-533-4372 (in California 800-356-2002).

FOREIGN NEWSSTAND DISTRIBUTOR: Worldwide Media Service Inc., 115 E. 23rd St., New York, New York 10010; 212-420-0588 FAX 212-420-1265.

Entire contents copyright ©1990 by M&T Publishing, Inc., unless otherwise noted on specific articles. All rights reserved.

The Audit Bureau

(continued from page 24)

possible is important for good performance, because Windows scans the whole table for every memory movement, which happens quite frequently. An additional step could be to detect when an entry at the end of the table is being freed, so *cwSegs* could be reduced, but that is probably more work than it's worth.

Once an empty entry is found, Windows is called to allocate the amount of memory requested. The *GMEM_MOVEABLE* flag is always used. If the application requires it, the function could be modified to accept the memory flags as an argument. If all goes well, the function returns a *pSeg*, a pointer to the entry in the table; zero returned indicates a failure.

void FAR PASCAL SegmentFree(PSEG pseg) — This function frees the data associated with the *pSeg* argument, and frees the table entry by placing it on the free list. This function should not be called on a fixed table entry, because it will make that entry available for general use.

void FAR PASCAL DataFree(PSEG pseg) — This function frees the data associated with the *pSeg* argument, but the *pSeg* itself is still reserved. Its table entry will contain zero. *SegmentRealloc* may be used later to allocate some memory to the *pSeg*.

BOOL FAR PASCAL SegmentRealloc(PSEG pseg, DWORD size) — This function changes the amount of memory allocated to a *pSeg*. It can be used on any valid *pSeg*, whether or not any data are currently allocated to it. This function is used for the initial allocation of fixed table entries (except *segDgroup*).

void FAR PASCAL SegmentError (void) — This function is present only when the *DEBUG* switch is on. It is the central exit point should any of the debugging checks fail. As written, it does nothing but *FatalExit(-1)*. The easiest way to use it is to simply put a breakpoint there, so you can do a stack backtrace as the first step to figuring out what went wrong. But it could also be spruced up to report an error without requiring a debugger to be present.

Next Month

By now, you should have a good understanding of the problems of managing multiple data segments under Windows, not to mention how a segment table can help you come to terms with these problems. Next month, we'll introduce a sample program, called SEGMENTS, that demonstrates the use of the *segtable* library and lets you watch as Windows updates the segment table before your eyes.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 89)

Vote for your favorite feature/article.
Circle Reader Service **No. 1.**

Three-Dimensional Graphics Using the X Window System

X can produce reasonably fast and visually acceptable 3-D graphics

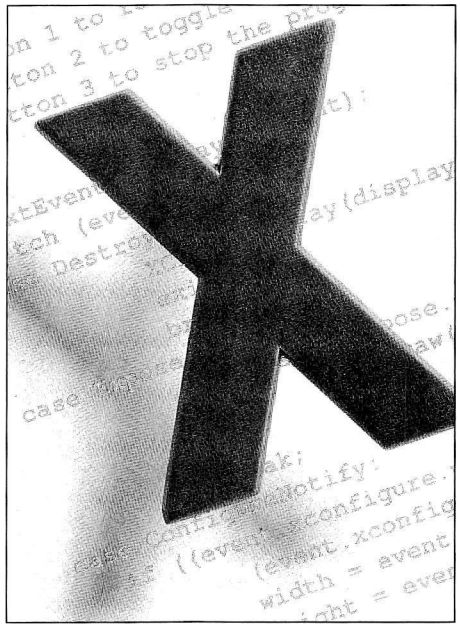
Michael Stroyan

The X Window System gives a new level of portability to graphics programs. X is available on many workstations. It is supported by X terminals, and by X server programs that can be run on PCs. There are several toolkits available to assist with the implementation of graphical user interfaces on X, but there is little support for three-dimensional graphics using X. This article discusses the options for creating 3-D graphics programs. It describes the experience of porting a 3-D graphics library to X, some of the issues that came up, and the solutions to those issues. An example program shows some of these solutions.

Approaches to Three-Dimensional Graphics

Three-dimensional graphics are created within X11 windows using a variety of approaches. These options include normal *Xlib* calls, 3-D graphics libraries (using either *Xlib* or a peer-level interface that cooperates with an X server to get directly to the hardware), and extensions to X11 that increase the server's capabilities. Each approach has advantages and disadvantages that make it appropriate in different circumstances.

Michael is a programmer with the Graphics Technology Division of Hewlett-Packard Company and can be reached at 3404 E. Harmony Rd., MS/73, Fort Collins, CO 80525.



Using the normal two-dimensional *Xlib* graphics to render transformed points is the most direct method. Conversion from 3-D to 2-D coordinates is done in a client program that uses *Xlib* commands to draw the 2-D results. These commands to the X server are either parts of the picture (such as vector and polygon primitives), or an entire picture (such as an *XPutImage* command). Using the X server to render vectors and polygons reduces the complexity of the task and improves program performance.

The rendering capabilities of the X server are limited, however. More sophisticated images can only be created by computing the complete image in a client program and sending that image to the server as an array of pixel values. When writing a program for use in X Windows, *Xlib* calls may be the best choice. Programs written for X Windows often use such features as X toolkits to take full advantage of the window system. Once tied to the window system by these features, there is little reason to use standard graphics libraries. The *Xlib* intrinsics are the only commonly available interface for graphics in X Windows. Putting all 3-D features in the program source allows (potential) portability of the program to all systems with *Xlib*. Using a 3-D graphics library limits the program to systems that have that library. The amount of extra effort involved to use only *Xlib* routines varies with the nature of the program. A CAD program may require a number of extra routines for transformation and input operations that a 3-D graphics standard library would have supplied. A ray tracing program, however, produces a list of pixel values that easily work with the *XPutImage* operation.

A slightly more difficult approach is implementing a 3-D graphics library that is layered above *Xlib*. Implementing a standard involves adapting the features of X to suit the required behavior of the standard. Matching all of

(continued from page 28)
the features requires additional work. This extra effort pays off because many existing programs that rely on this standard can then be used in X Windows. A few such libraries have been created. Hewlett-Packard has added *Xlib*-based X11 support to the HP Starbase and HP Graphical Kernel System (GKS) libraries. An X11 implementation of the GKS standard has been created at the University of Illinois at Urbana-Champaign. Template Graphics has produced an *Xlib* implementation for their Figaro PHIGS library.

A peer-level library has a very different relationship to the X server. This type of library arranges direct access to the display either by using X server extensions, or by using the low-level resource allocation utilities also used by the X server. The direct hardware access approach gives a library superior speed and broader use of hardware capabilities than an *Xlib*-based implementation. This is especially important for graphics workstations that support operations such as transforms, hidden surface removal, and shading in specialized hardware. A peer-level library sacrifices the networking capability of X. Programs using such a library must run on the same system that they display their output on. Hewlett-Packard uses this type of library for the Starbase and GKS libraries on HP-UX, and for the Domain graphics libraries on HP/Apollo workstations.

The X11 protocol has a mechanism for defining new groups of features, called "extensions." An X server implementation could provide an extension that processes and renders three-dimensional data. The MIT X Consortium is sponsoring a 3-D graphics extension named PEX (PHIGS Extensions to X). The PEX extension will provide support for PHIGS in the X protocol. A server supporting PEX will be able to render PHIGS primitives. The server will also have the option of providing support for storing groups of primitives in the server. This will allow complicated images to be edited and redrawn without retransmitting all the data from the client program to the display server. A sample implementation of PEX should be ready sometime near December of 1990. It will take additional time for vendors to provide efficient PEX implementations. Vendors of X servers on limited hardware, such as X terminals, may never choose to provide the PEX extensions.

The Approach Taken

Hewlett-Packard has produced several libraries to assist with the creation of 3-D graphics in the X Window System. These libraries use both the peer-level interface approach and the layered *Xlib* approach for Starbase and GKS libraries. Prototypes of the PEX extension approach were demonstrated at the SIGGRAPH '88 and SIGGRAPH '89 conferences. The rest of this article concentrates on implementing a device driver to layer the Starbase graphics library above the *Xlib* library. The techniques developed to implement this "Starbase on X11," or "sox11" driver, apply to creating other libraries and programs that display 3-D graphics using *Xlib*.

Transformations and Clipping

Transforming 3-D coordinates to 2-D coordinates is a well-understood process. Matrix operations are applied to the 3-D coordinates to perform rotations, translations, and perspective projections. These techniques are described in many computer graphics reference books, including those listed in the bibliography. The sox11 driver relies on existing utilities in the Starbase library to perform transformations and clipping.

The program in Listing One (page 92) uses three matrices to process coordinates through modeling, viewing, and device scaling transformations. The modeling matrix is made of a concatenation of rotations about the X and Y axes. The matrix is updated to follow mouse movements. The viewing matrix is precomputed to show a perspective view looking at the origin, from a point at (0, 0, – 15). It maps the world coordinates into virtual device coordinates (VDCs), ranging from (0, 0, 0) to (2, 2, 2). The device matrix maps VDC units to device coordinates — the window coordinates of *Xlib*.

The VDC to device coordinate transformation is slightly different for X than for most other devices because the size of the X Window "device" can change. The sox11 driver leaves the device coordinate transformation unchanged when a window is resized. If a window shrinks, the driver shows a "porthole" into an image larger than the window. An application programmer can watch for resize events and update device mapping to match the new window size. This gives a "rubber sheet" effect — the image stretches to match the size of the window. In the program in Listing One, the device transformation matrix is updated whenever the window size changes so that the image maps to the new window size. There is no clipping code in this example; it relies on the object being within the viewing volume.

(continued from page 30)

Partial Polygons

To improve performance, sox11 uses X server polygon drawing routines to draw polygons. This causes a problem with some polygons. The Starbase graphics library supports polygons with holes in them by defining primitives called partial polygons. (The PHIGS graphics standard calls these groups of polygons "Fill Area Sets.") The program in Listing One shows a cube with a hole in one side. The hole in the face of the cube is a partial polygon. Polygons with holes in them are described as a group of partial polygons, followed by a normal polygon. Each polygon or partial polygon is a loop of vertices. These loops may be the outside edges of separate areas, or the outside and inside edges of a single polygon. *Xlib* does not explicitly support holes in polygons. To use *Xlib* polygon routines and produce the desired result, the sox11 driver rearranges the vertices and relies on a detail of the definition of *Xlib* polygons.

X protocol defines that drawing a polygon only affects the pixels that are within the interior, or on an edge other than the right and bottom edges. This allows adjacent polygons to share common edges without both polygons affecting the pixels on the common edges. A mesh of vertices drawn with an exclusive OR won't leave gaps between polygons. As a result, if a polygon is pinched down so that two edges are on the same coordinates, no pixels are changed along the pinched area. All of the pixels are on a right or bottom edge and are therefore left alone. If such a double edge crosses the interior of a polygon, there is no gap where the edges cross the interior.

When asked to draw a complex polygon, the sox11 driver connects the partial polygons to each other by one of these thin double edges. The result is that one loop of vertices drawn by *XFillPolygon* appears to be multiple disjointed loops. If the polygon is edged, the edges are later drawn along the edges of the partial polygon loops with an *XDrawLines* call for each smaller loop. Figure 1 shows an example polygon. This polygon is defined to Starbase as a partial polygon, with the vertices numbered 1, 2, 3, 4, and 5; followed by a polygon with vertices numbered 6, 7, 8, 9, and 10. The X server is sent one *XFillPolygon* request with all of the vertices from 1 to 11. Then the edges are sent as two *XDrawLines* calls with the vertices of the Starbase partial polygon and polygon calls.

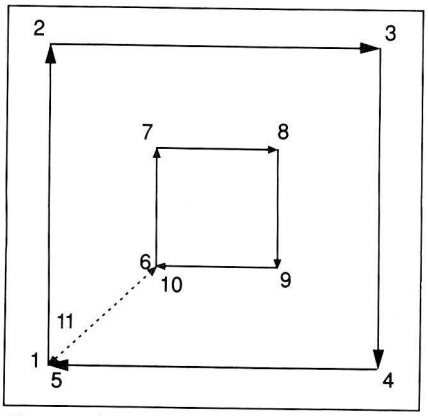


Figure 1: *Drawing a polygon with a hole in it*

Color Map Support

X protocol deals with many different types of hardware, each of which has a variety of methods for representing and controlling colors. X protocol categorizes the representation of colors into six different visual classes: *StaticGray, GrayScale, StaticColor, PseudoColor, TrueColor,* and *DirectColor.* These classes indicate whether a display interprets pixel values as levels of brightness or as indices into tables of colors. They also indicate whether any color tables have writable entries, and whether the bits in a pixel are grouped into particular bits that determine the red, green, and blue brightness of a color. Some displays, such as the HP Turbo SRX, support multiple visual classes at the same time. The sox11 driver supports all six classes of visuals, but I will only discuss the *PseudoColor* class, because it raises the most interesting issues. A *PseudoColor* class visual indicates that pixel values are indices into a writable table of colors, called a "color map."

The Starbase library, like most graphics standards, is based on a model of controlling a virtual device. One result is that the library presents color maps as being completely available to application programs. The *Xlib* concept of color maps is different. *Xlib* knows that many programs will be executing on one display, and usually must share a single hardware color map. Reconciling these two approaches to color maps can be done in two different ways. Either the library's color indices can be mapped to a different set of allocated *Xlib* indices, or a new software color map can be allocated by *Xlib* for the exclusive use of the sox11 driver.

The allocation of individual indices is unsatisfactory if an application uses Boolean pixel changing functions to combine colors on the display. A program may expect to OR together a red color in index 1 and a green color in index 2 to produce a yellow color that

(continued from page 32)
it set in index 3. Rearranged color indices will not necessarily produce the expected result.

Allocating a complete color map also has drawbacks. Because most displays only support one color map at a time, the server switches the hardware color map to correctly display the current window. All other windows are then displayed with essentially random colors. This can be visually jarring, making the rest of the windows more eye catching than the supposed center of interest.

The sox11 driver uses some of each color method to avoid most of the weaknesses of each method. The driver starts by using the default X color map. It then allocates a new color map if a program changes any of the color map entries. This means a Starbase program can share the common default color map, as long as it doesn't care about the index values for each color. When a program needs to control the color of particular indices, a full color map is used, which gives the program complete control.

When a color map is allocated, sox11 uses some color map tricks to replace missing *Xlib* features. The Starbase library defines a control called "display enable." This selects the bits in a pixel that are used to index into a color map. When looking up a color, those bits that are not display enabled are read as zero. Several HP displays provide this control in hardware, but the X protocol does not have this feature. The sox11 driver simulates hardware display enable by rewriting the color map. Thus all indices that differ only by bits in display disabled planes have the same color.

The display enable feature can be used for double buffering. In double buffering, bits in a pixel are divided into two sets. Half of the bits hold the image that is being displayed, while the other half are cleared and redrawn to create a new image. With double buffering, the transition from displaying one image to displaying the next is extremely fast. When the undisplayed image is ready, the color map is rewritten and the new image is displayed in completed form. There is no flicker while the image is changing. This is important for animation, where images are being constantly updated.

Listings Two (page 98) and Three (page 98) show the header declarations and code for an *Xlib* double buffering utility. This allocates colors from a shared color map, then maps the program's pixel indices to the values returned from *Xlib*. The utility will only work with *PseudoColor* class visuals, and will only succeed if there are sufficient free pixels to allocate from the color map.

Hidden Surface Removal

When drawing 3-D objects as wireframe outlines, removal of hidden lines is helpful, but not vital, to the perception of the objects' structures. Perspective projection and the motion of vertices during rotations give clues about the distance of points. When drawing objects with solid faces, removal of surfaces appearing behind other surfaces is very important. The image only makes sense to our eyes if "hidden surfaces" remain hidden.

The sox11 driver does not perform hidden surface removal. An application program using either *Xlib* or the sox11 driver can do some hidden surface removal before drawing polygons. A program can detect and remove the faces of objects that are directed away from the viewer. A vector pointing perpendicularly out from the face (a "normal"), is either entered as part of the object data, or computed from the vertices of each face. Perspective transformation is applied to this normal vector. If the resulting vector has a negative Z component, then the face is on the back of the object as seen from this

(continued from page 34)
viewing position. The back faces can be removed, because they should never be visible. Back face removal detects all hidden surfaces of a single convex object, but won't always be sufficient for multiple or more complicated objects.

Another hidden surface removal method that can be applied by an application program using the sox11 driver is called the "painter's algorithm." This consists of sorting the polygons in an object so that the most distant are drawn first. When two polygons overlap each other on the display, the closer polygon will be drawn last and will cover up the more distant one. The faces of the object drawn by the example program in Listing One have been sorted so that the painter's algorithm applies to those faces that are not back faces. For more general objects, an actual sorting of the faces is required.

The painter's algorithm fails if faces pierce through each other or if a group of faces are arranged to form a cycle of overlapping polygons. If face A hides part of face B, face B hides part of face C, and face C hides part of face A, the polygons cannot be sorted. Another technique must be used. Two possible methods are scan line hidden surface removal or Z buffering. To use either of these techniques with *Xlib* requires that the conversion from vectors and polygons to scan lines or pixels be done by the client program, instead of the X server. If an application must handle graphics of such complexity, an approach different than layering on *Xlib* may have to be used. Peer-level interface Starbase drivers perform these hidden surface removal operations quickly, using hardware scan conversion and Z buffering.

Summary

The X Window System produces reasonably fast and visually acceptable 3-D graphics. Depending on application requirements, the existing *Xlib* graphics interface, extensions to the X protocol, or coexisting peer-level 3-D graphics libraries can be used to provide a variety of capabilities, performance ranges, and portability levels. *Xlib* calls, which provide the best portability among the currently available X implementations, can be used to implement most features of existing 3-D graphics libraries.

Bibliography

Foley, J.D., and Van Dam, A. *Fundamentals of Interactive Computer Graphics*, Reading, Mass.: Addison-Wesley, 1982.

Jones, O. *Introduction to the X Window System*, Englewood Cliffs, N.J.: Prentice-Hall, 1989.

Newman, W.M., and Sproull, R.F. *Principles of Interactive Computer Graphics*, New York, N.Y.: McGraw-Hill, 1973.

Nye, A. *Xlib Programming Manual for Version 11*, Newton, Mass.: O'Reilly and Associates, 1988.

Rogers, D. F., and Adams, J. A. *Mathematical Elements For Computer Graphics* New York, N.Y.: McGraw-Hill, 1976.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 92.)

Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

Pick-A-Number Interfaces

Just say "no" to window dressing

Bob Canup

In this age of "user friendly" interfaces with pop-up windows, pull-down menus, and a plethora of moving bars in almost every program, you may question why I would write an article about old-fashioned, pick-a-number menus. The answer to that puzzle is that the pick-a-number menu has some powerful subtleties that make it difficult to replace in certain computer applications. Pick-a-number interfaces are the easiest interface for a programmer to write into a program, and they consume the least amount of computer resources. If pick-a-number interfaces are properly coded, people who have no computer experience can use them easily. In addition, these interfaces allow touch typists to work without moving their eyes away from the screen.

I recently had the misfortune to enter a large amount of data into a mailing list program that has all of the modern interface fads — exploding windows, function keys, moving-bar menus, sound effects, and Yes/No confirmations (which, amusingly, used function keys, instead of the Y or N keys). The programmer obviously had devoted a great deal of time and effort to the interface portion of the program. The reason that I said "misfortune" earlier is that in any programming project to which a given amount of effort is made, the effort expended in glitter and gloss comes at the expense of substance and functionality. With the program in question, the missing functionality was the lack of a means to handle the possibility of a power failure. If a power failure occurred while you were adding data to the mailing list, all of the data in the mailing list would be lost. In addition, if you exited the program without manually closing the database, all of the data that was ever entered into the program would be lost. The only way to back up the data was to use the user manual to close the file, and then copy the data file to a different file name.

Now I'm not about to claim that the addition of a pick-a-number interface to this program would have solved the program's problems. My earlier comment about the trade-offs that can be made when the level of effort is fixed is true, but it is also true that all too often, the level of effort is not fixed. Given an easier interface to write, the programmer simply lowers the level of effort that is devoted to the program. Nevertheless, programmers who change to a simpler user interface gain more time to at least consider what they are doing with the rest of the program.

Programmers write complicated user interfaces for a number of reasons. For one thing, a fancier interface is more challenging to write, so many programmers find it more interesting to code. A three-ring-circus-style interface is useful for impressing compu-boobs, and may, in some cases, sell lots of programs. Most programmers don't understand what user friendly means, so they assume that it means something that dazzles the eye. Finally, a trendy, faddish program interface makes the software look modern and up-to-date.

Pick-a-number interfaces are appropriate in the following circumstances: 1. You are writing a custom program for a business, and the program will be used by people who are interested in the program's utility, not in how flashy the program appears; 2. The code will be used by many people who may not be sophisticated computer users; 3. The amount of time needed to build the finished program is important; 4. Computer resources are limited; 5. Speed and error prevention are important in the program.

Pick-a-number interfaces are not appropriate in the following cases: 1. The program is intended for use by management personnel who fancy themselves as "computer literati" and who request other interfaces; 2. The program is game software, where glitter and gloss is the whole purpose; 3. You are trying to impress someone with your programming skill; 4. Your pro-

(continued on page 41)

Bob designed the first 5-MHz CPU card for the S-100 bus (TEI 1978). He also designed the TEI System 48, the Maxicom DL, and authored the shareware program Onbase. He may be reached c/o Blackbelt Software, P.O. Box 31075, Houston, TX 77035.

(continued from page 38)
gram really doesn't do much, so it is necessary to cover up this fact; 5. You are developing a program for commercial distribution, and the marketing people insist upon a different interface.

The 80/20 Percent Rule

Granted, it is possible to write a clumsy user interface in any technology. Despite the fact that the interface of the mailing list program mentioned earlier offered all of the leading-edge ways of doing things, the interface was awkward and slow to use. I generally had to step through three levels of moving bars in order to do what I wanted to do — just enter data and print mailing lists. Of course, this awkwardness is also true of most pick-a-number interfaces. Generally, programmers require users to wade through pages of menus before the users are allowed to do what they want to do. The reason for this phenomenon is that it is just as difficult to write code that is rarely used in the program as it is to write code that will be used often. As a result, programmers give equal importance in the user interface to both often-used and rarely used options.

When the term "user friendly" is applied to a program, it means that the program usually does just what the user wants it to do with a minimum amount of fuss and bother on the user's part. The application of the "80/20 percent rule" to menu design is an important part of creating a user-friendly program. You should determine what will be done with the program 80 percent of the time, and then devote menu options that accomplish exactly these functions to the 20 percent of the code that will perform them. Then place the most-used options at the front of the menu. The menu shown in Example 1 illustrates this approach. (The code that generates this menu is shown in Listing One, page 100.) Most of the time, the user will pick either option 1 or option 2, which handle the primary functions of the mailing list program.

```
                Mailing List Menu
        ( Press Esc key to leave program)

   1. Enter new mail list data
   2. Print a Zip code sorted mail list
   3. Change existing data
   4. Delete a single address from the list
   5. Browse through the existing data
   6. Backup data
   7. Print a mail list not sorted by Zip
   8. Perform other functions


   Enter the number of your selection and
   press Enter key:
```

Example 1: *In pick-a-number interfaces, the most commonly used options should be available early in the menu*

This 80/20 design rule leads to a program that users usually enjoy working with because it usually does just what they want to do with a minimum amount of bother. Also, following this rule results in a program that requires a minimum amount of training for the people who use it. I have found it far easier to teach people how to use a program designed in this fashion, than to show them how to use a bell-and-whistle-style interface. Virtually anyone who can read can learn how to use a program that is designed according to the 80/20 rules. I maintain that successfully learning how to use a program that is driven by a moving bar or by function keys requires a bit more computer experience.

One interesting aspect of the menu in Example 1 is the requirement that the user must press the Enter key after a menu selection is entered. When I first began coding menus into programs, I designed the programs so that the requested function was performed without requiring the user to press the Enter key. This approach limited the size of the menu to only ten functions (0. .9). In addition, the users were uncertain about whether or not the computer was active.

The issue of consistency in data input is also related to this Enter key issue. I have found it vital to always require that input into a data field be followed by a press of the Enter key. If you allow a user to skip automatically to the next field when a field fills up, but you require the user to press the Enter key only if the field is not filled, then the preprocess of data entry requires constant, conscious checking on the part of the person who enters the data. Additionally, the cursor can be placed in the wrong data field if the user tries to enter more data than a field can hold.

Data entry can never become a subconscious, automatic task because the speed of data entry is limited. Subconscious tasks are executed much more quickly by the user than are conscious tasks, and the limited data entry speed places a higher level of stress on the data entry personnel. Imagine the stress that would occur during the process of driving if stopping the car sometimes required you to press the brake pedal, and other times (if, for example, the car in front of you weighed more than 3,800 pounds) you had to twist the rearview mirror without pressing the brake pedal. It may appear wasteful to always require the user to press the Enter key after data entry in each field is finished, but the technique both substantially speeds up the data entry process and increases the program's ease of use.

Pick-A-Number Code

I am basically a lazy programmer who wants the computer to do as much of the grunt work as possible, so I wrote code that automates the process of centering and dividing menu data into columns, and provides error rejection during user input. (As Robert Heinlein said, "Progress is not made by early risers, it is made by lazy people looking for an easier way of doing something.") The code that implements these functions is written in Modula-2 and has been tested with the Logitech Modula-2 compiler (Version 3.0). The process of converting this code so that it can run with other Modula compilers will probably be straightforward.

The procedure Grab in the module READA (Listings Two and Three, page 100) is a general-purpose data-field entry routine that requires a press of the Enter key after a data field is filled. An alarm sounds if an attempt to overflow the field is made. The procedure Menu in the module MENU (Listings Four and Five, pages 101 and 102) implements pick-a-number interfaces for menus that contain up to 60 entries. This procedure centers the menu items on the screen, and divides the menu entries so that they occupy from one to four columns for display.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 100.)

Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

Self-Adjusting Data Structures

Use self-adjusting heuristics to improve the performance of your applications

Andrew M. Liao

Application programs are often developed using standard data structure techniques such as stacks, queues, and balanced trees with the goal of limiting worst case performance. Such programs, however, normally carry out many operations on a given data structure. This means that you may be able to trade off the individual worst case cost of each operation for that of the worst case cost over a sequence of operations. In other words, any one particular operation may be slow, but the average time over a sufficiently large number of operations is fast. This is an intuitive definition of amortized time, a way of measuring the complexity of an algorithm. In this case, the algorithms to be concerned with are those that carry out operations on data structures.

The heuristic I'll discuss in this article is called the "structural self-adjusting heuristic." To illustrate what I mean by self-adjusting, consider the following example: Suppose you're running an information warehouse and your task is to distribute information to people who request it. The information in this warehouse could be stored in a fixed order, such as the order of information in a library. You quickly notice, however, that certain pieces of information are requested more often than others. You could make the job easier by moving the most often requested information close to the service counter. This means that instead of having to search through the depths of the warehouse at any given time, you have a good portion of the most requested information nearby.

As this example suggests, self-adjusting heuristic algorithms are ideally suited to lists, binary search trees, and priority queues (heaps). In lists, the heuristic attempts to keep the most frequently accessed items as close to the front as possible. In binary search trees, the heuristic attempts to keep the most frequently accessed items as close to the root as possible, while preserving the symmetric ordering. Finally, in heaps, the heuristic attempts to minimize the cost of modifying the structure, and partially orders the heap in a simple, uniform way. To illustrate how these algorithms can be implemented, I've provided sample Pascal source code.

Andrew received his master's degree in computer science from RPI in Troy, New York. He can be reached through his Bitnet address, which is aliao%eagle @wesleyan.bitnet. You can also reach him through the DDJ *office.*

Self-Adjusting Lists

A singly linked list is a group of records where each record contains one field that holds an individual piece of user data, and another field that holds a pointer to the next record in the list. An initial pointer that indicates which record starts the list is (or should be) kept. This pointer enables you to search, insert, and delete operations.

Move-to-Front Singly Linked Lists

To understand how the move-to-front (MTF) approach works, consider a situation in which a particular application uses an open hash table with a linked list that is associated with each array location. Suppose that the hashing routine for this application is as good as it can possibly be. If you wish to improve the search performance without unduly complicating the supporting code, however, you might examine the performance of the search performed on the lists. Chances are that certain elements are accessed more often than others. The use of either a transpose or a frequency count heuristic (two

(continued from page 44)
other common access approaches) does not appear to be a good idea because of the search overhead involved with each approach. Both methods require either a local exchange operation or extra searching in order to reinsert an accessed item into the correct part of the list. Also, the count method requires a change in the list: The addition of an integer field that maintains the access count. All three heuristics are effective in that they search less than half the list.

One reason why the MTF heuristic performs better than the transpose method is that the transpose heuristic causes the list to converge more slowly to its optimal ordering. In the case of MTF, an element is brought to the front of the list. Furthermore, such an element quickly "sinks" to the end of the list over the course of a sequence of accesses if that element is not a sufficiently wanted item. Essentially, MTF may be viewed as an optimistic method in the sense that the method "believes" that an accessed item will be accessed again. Analogously, the transpose heuristic may be viewed as pessimistic in that it "doubts" that an accessed item will be accessed again. The count method is a compromise between the two.

As Figure 1 and Listing One (page 105) illustrate, searching is the key operation for the MTF heuristic. This search operation is very much like a normal search on a singly linked list, except that an extra pointer is kept to the current predecessor of the list node that is currently being examined. Once a given item is found, the pointer to its predecessor node is used to alter the predecessor node's link field so that the link field points to the successor node of the accessed item. The link field in the desired node is then altered to point to the first element in the list, and the head-of-list pointer is set to point to the new front-of-the-list item. For all intents and purposes, the insert operation is a push operation — the new item is immediately put at the front of the list. Finally, an MTF search is used to perform a delete. If the item to be deleted is located at the front of the list, that item is removed from the front of the list.

Self-Adjusting Heaps

A "heap" is a tree-based data structure in which each record node keeps a key, along with pointers to the record node's successors. The heap maintains an ordering of items such that every node has a key less than, or equal to, the keys of the node's successors. This last description is the concept of "heap-ordering."

There are a number of classical priority queue structures (such as 2 – 3 trees, leftist heaps, and binomial heaps) that are amenable to fast merging operations. Of these, the simplest scheme for maintaining a heap with fast merge operations is the "leftist heap," which was developed to maintain partially ordered lists with a logarithmic time-merge operation. A leftist heap is based upon a binary tree node that contains rank, weight, and two pointer fields (to indicate left and right children). The rank field is defined to be 0 if a node is a leaf. Otherwise, the rank field is defined as one more than the minimum value of both the rank of the leftchild and the rank of the rightchild.

A binary tree is a leftist heap if it is heap-ordered and if the rank of a given leftchild is greater than, or equal to, the rank of its rightchild sibling. The problem with maintaining leftist heaps is that the configuration of the data structure is based upon the rank definition. All operations are heavily dependent upon the value kept in the rank field of a given node. To illustrate the point, I'll describe the leftist heap merge operation.

The leftist heap merge operation is made possible by a modified version of an "enqueue" process (which takes a heap and a queue pointer's record as parameters). This particular enqueue operation saves the root of the heap

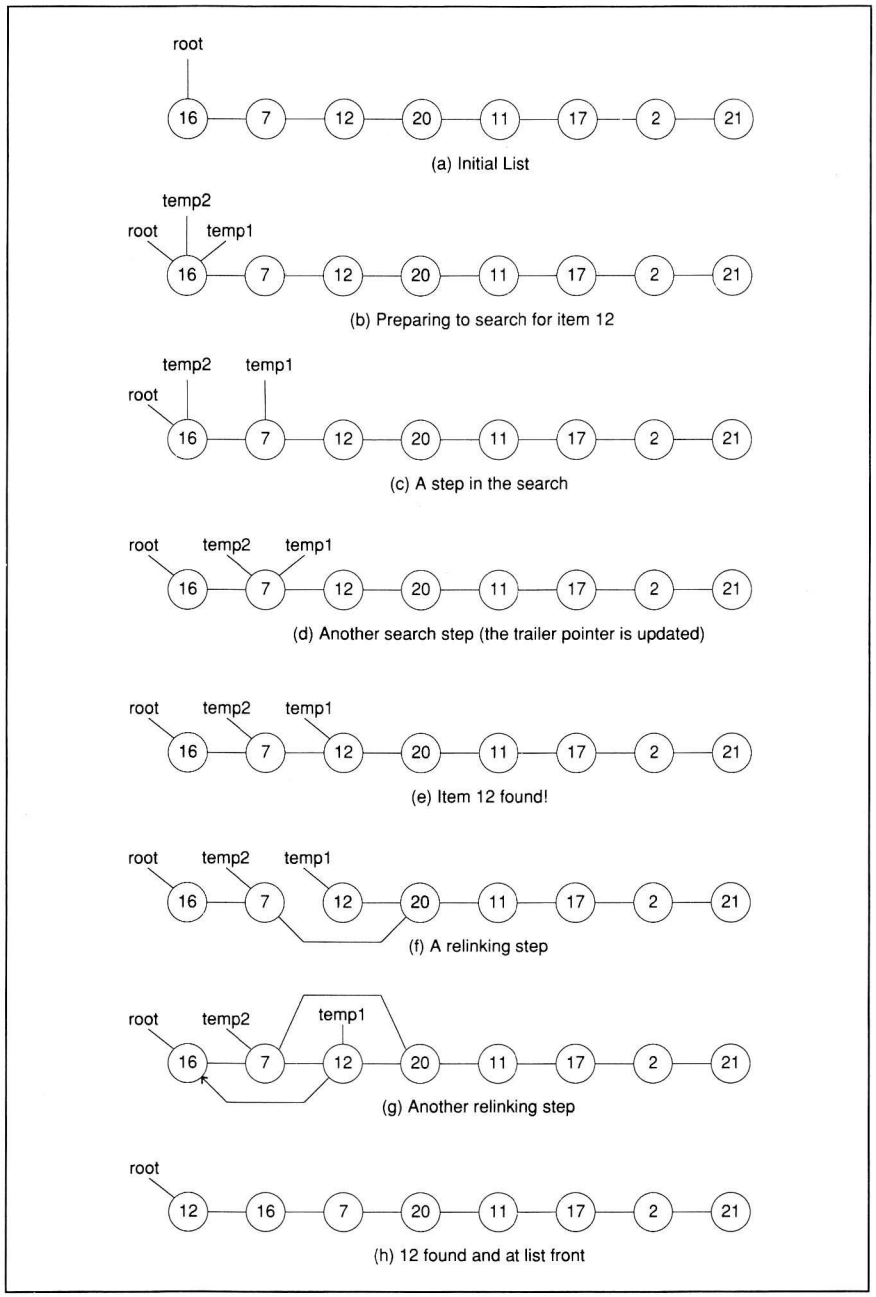


Figure 1: *Finding Item #12 in a list with the MTF restructuring heuristic*

(continued from page 46)
and moves the front queue pointer down to the rightchild of the root just saved. You then break the link between the saved root and its rightchild. If the queue pointers are both empty, point the front and rear pointers to the root node that was just saved, and set the rightchild pointer field of the root to empty. Otherwise, point the rightchild pointer to the node currently pointed to by the rear queue pointer, and point the rear queue pointer to this newly obtained node.

Implement the merge with the following steps: While neither of the two heaps being merged is empty, call *enqueue* with the currently minimum key and with the queue pointer's record. Next, while the "first" heap is not empty, call *enqueue* with that heap and with the queue pointer's record. Perform the same steps for the other heap. These three processes merge the right path. Complete the process with a bottom-up traversal of the right path in order to fix up the rank fields and to perform any necessary swaps to maintain the structural invariant.

Now point to the current two bottommost nodes on the merge path. If there is no left sibling of the bottommost node, make the rightchild a leftchild and set its parent's rank field to 0. Next, set the rightchild's rightchild pointer field to empty. If the bottommost node has a left sibling, compare the two children and swap them when the rank of the left sibling is less than that of the right sibling. In any event (given this case), set the parent's rank field to 1+ rank of the rightchild. Also note which nodes are the next two bottommost nodes on the merge path at this point, and make sure that the parent node before this step points to the rightchild. This process continues until the root is reached when the root of the new heap is returned. Once the merge operation for leftist heaps has been described, the other heap operations are easy to implement.

This description of the merge operation suggests that two passes are required over the merge path. The question remains: How do you improve performance without unduly complicating the algorithms that maintain the heap? This can be done with a restructuring method that essentially exchanges every node on the result heap's right path with the node's left sibling. The version of the technique presented here also has a feature in which one top-down pass completes the merge. The resulting structure, called a "top-down-skew heap," is a self-adjusting analog of the leftist heap.

Top-Down-Skew Heaps

A skew heap is based upon a simple binary tree node that contains a weight plus pointer fields to left and right children. The process of merging is made possible by another modified version of the enqueue algorithm. In this case, it's not necessary to maintain the rank/balance field in order to obtain logarithmic, amortized performance.

This particular enqueue operation saves the root of the heap and moves the front queue pointer down to the rightchild of the root just saved. You then break the link between the saved root and its rightchild by changing the current leftchild into a rightchild. If the queue pointers are both empty, point the front and rear pointers to the root node just saved, and set the root's leftchild pointer to empty. Otherwise, the newly obtained node becomes the leftchild of the node that is currently indicated by the rear queue pointer, after which the rear queue pointer is changed to indicate the newly obtained node. (See Figure 2.)

The following steps implement the merge: While neither of the two heaps being merged is empty, call *enqueue*

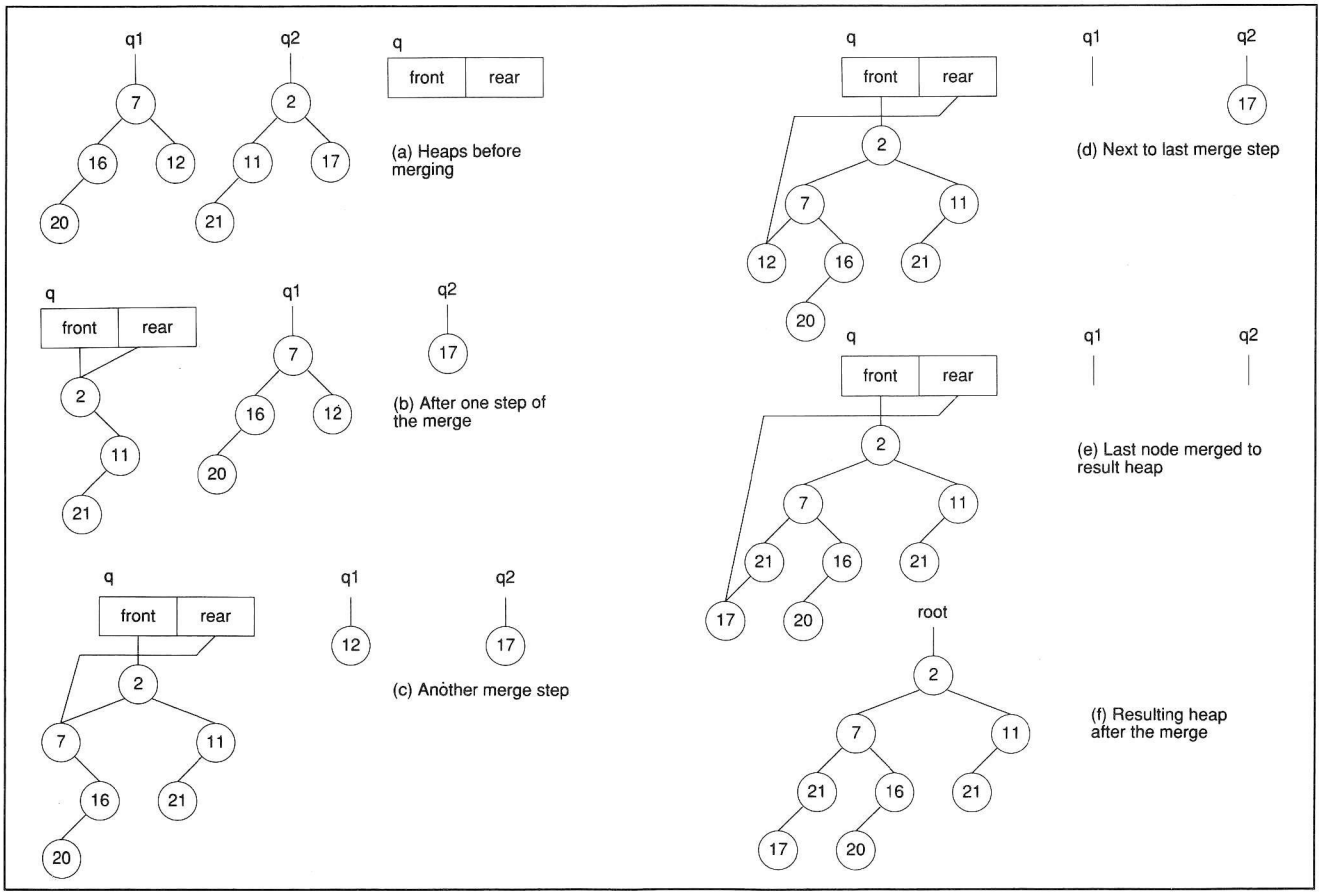


Figure 2: *Merging two skew heaps*

(continued from page 48)
with the heap that contains the current minimum key and the queue pointer's record. Next, while the "first" heap is not empty, call *enqueue* with that heap and with the queue pointer's record. Follow the same process for the other heap. (This approach is analogous to Tarjan and Sleator's conceptual noting that the left and right children of every node on the merge path are swapped. The implementation used here, however, is a variation.) Once either of the two heaps being merged becomes empty, merely attach the remaining heap to the bottom of the result heap's left path. Again, the rest of the heap operations are easy to define.

Pairing Heaps

Much like the leftist heap, the binomial heap has an analogous self-adjusting counterpart. This new structure, called the "pairing heap," is a recent development in heaps that supports the *decreaseKey* operation. The essential definition of the pairing heap, like that of the skew heap, is based upon a simple binary tree record node that contains at least weight plus three pointer fields (to indicate the parent and the left and right siblings). Like most heaps, the pairing heap depends upon a merge operation, but has a less complicated scheme than its classical counterpart.

In the case of the binomial heap, you need to maintain a forest of trees where each tree contains a number of nodes equal to a non-negative integer power of two. Thus, a binomial heap of n items can be represented by a forest in which each tree corresponds one-to-one with the one bit that represents the value of n in binary. (This eventually leads to the fact that all of the binomial heap operations are, in the worst case, logarithmic time.) Needless to say, the code needed to implement a binomial heap merge operation is complicated and difficult to maintain.

The merge operation for pairing heaps begins by determining which of the two heaps has the minimal weight at the root. The heap with the non-minimum key at the root then becomes the child at the root of the other heap. The heap that is being made into a subtree points its root node right sibling pointer to the child of the root of the other heap. Furthermore, the first heap's parent pointer is set to the new heap root, and the new heap root points its leftchild pointer to the root of the heap that is being made into a subtree. The merge operation returns the root of the new heap. (See Figure 3.)

Given the above definition of the merge operation, the *DeleteMin* operation (see Figure 4 and Listing Three, page 106) is easy to describe. I will describe the front-back one-pass variation here. To begin, save the root node and keep a pointer to the leftchild. Next, empty the pointer to the root. While subtrees are linked to the leftchild of the root, remove trees in pairs (beginning with the leftchild) and merge the trees, then merge the result to the heap pointed to by the root pointer. Repeat this step until there are no more trees. (The pairing heap derives its name from the restructuring operation that takes place during a *DeleteMin*.)

Describing the *DecreaseKey* operation for pairing heaps (see Figure 5) is just as easy. This operation assumes that you have direct access to the node

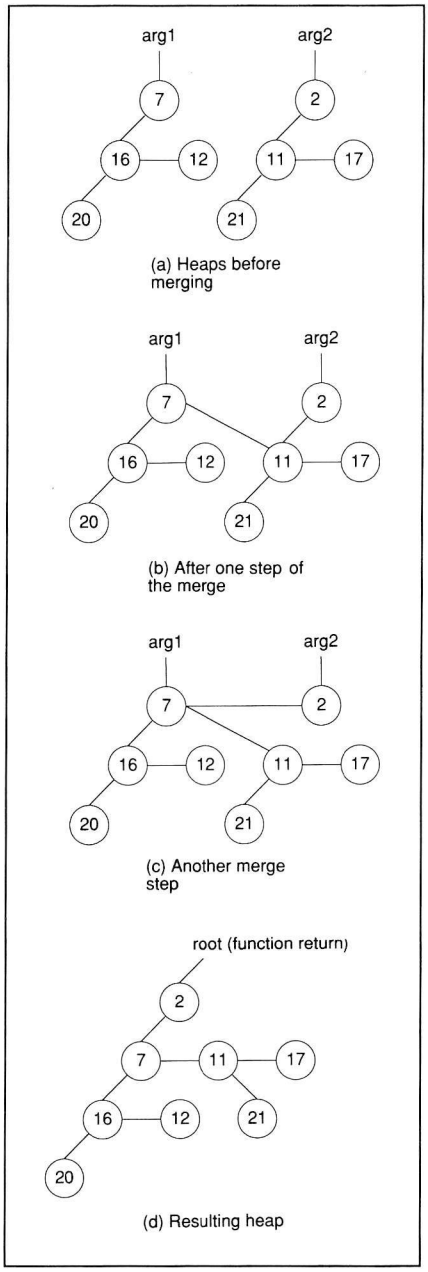


Figure 3: *Merging two pairing heaps*

whose weight field is being decreased, to a root to the heap that contains the node, and to the value by which you wish to decrease the weight. Go to the parent of the node that is being operated on, and then go to the leftchild of that parent. Scan along the right sibling list to find the predecessor of the node that will be operated upon. When the predecessor is located, clip out the tree rooted at the node upon which you wish to carry out the actual *Decrease-Key* operation. To clip out the tree, link around the node in question. If the node is a leftchild, make its right sibling the new leftchild. Now decrease the weight and merge the tree that is rooted at the node with the root of the pairing heap.

The simple local restructuring heuristics presented here provide an elegant approach to the development of heap structures. In fact, these heaps are simpler to understand and to implement than either the leftist or binomial heaps. Furthermore, indications are that self-adjusting heaps are just as competitive in practice as their classical counterparts. In any event, I've presented two very different (though effective) local restructuring heuristics. The first heuristic reorganizes lists in order to make frequently requested list items more accessible. The second heuristic applies a simple local restructuring method (in place of maintaining balance/accounting data and resolving special structural cases) in order to quickly maintain both the structure and the partial ordering of a heap.

Self-adjusting heuristic algorithms are ideally suited to be used with lists, binary search trees, and priority queues (heaps)

Now let's consider an efficient self-adjusting heuristic for binary search trees. This algorithm makes frequently requested items in the tree more easily accessible, and quickly maintains both the structure and the sorted ordering of the tree.

Self-Adjusting Binary Search Trees

In a "binary search tree," each node keeps a key along with two pointers to the node's successors. The ordering is such that if a node has key *K*, every node in that node's left subtree must have keys less than *K*, and every node in its right subtree must have keys greater than *K*. This is known as "symmetric ordering." The performance costs of generic binary search tree operations are, in the worst case, logarithmic time (if the input data is sufficiently random). Such a tree may also degenerate as a result of insertions and deletions, and yield steadily poorer performance.

The process of tree degeneration has led to the development of various height/weight balanced trees and B-tree schemes. Although these various schemes guarantee logarithmic worst case times per operation, some of the schemes are not as efficient as possible

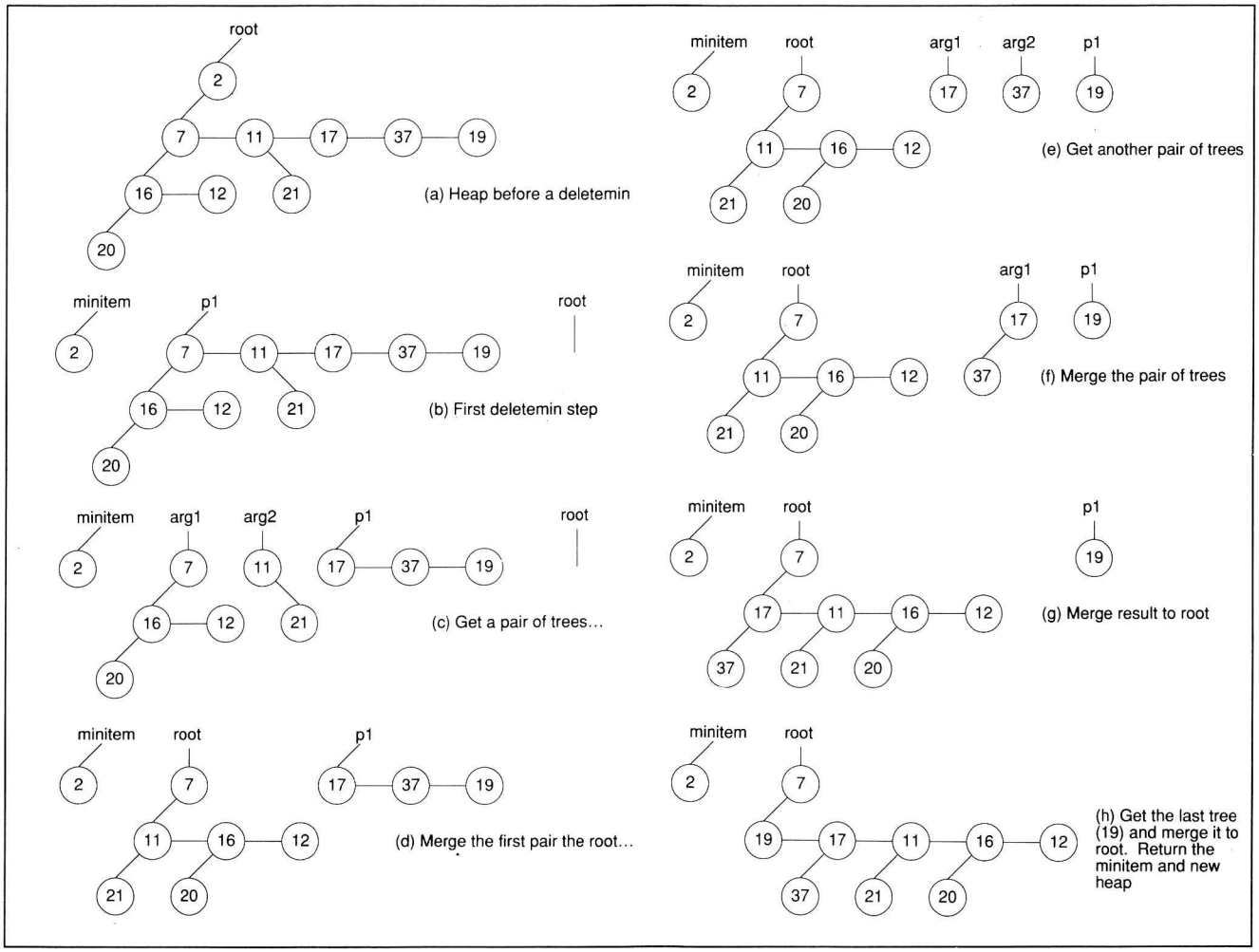


Figure 4: DeleteMin *process on a pairing heap*

under nonuniform access patterns. Furthermore, many of these schemes require extra space for balance/accounting information, and the need to keep this information current tends to complicate maintenance of the data structure. Certain cases must be checked on each update, thus incurring a large overhead.

Rotation is the key technique that makes some of the balanced and previous self-adjusting tree schemes possible. In fact, rotation plays a part in the implementation of the splay tree. Before this discussion continues, it is necessary to understand how a right rotation and a left rotation at any node of a binary tree are performed.

As Listing Two (page 105) shows, you implement a right rotation with the following steps: If the pointer to a starting root of some tree is not empty and that node has a left subtree, save the pointer to the root and then save the pointer to the right subtree of the initial left subtree. Then make the pointer to the initial left subtree the new starting root pointer, and let the original root be the rightchild of the new root. Finally, designate the pointer to the saved rightchild of the original leftchild as a leftchild of the new root's rightchild (which is the original root).

Implement a left rotation in a similar manner. If the pointer to a starting root of some tree isn't empty, and that node has a right subtree, save the pointer to the root and then save the pointer to the left subtree of the initial right subtree. Then, designate the pointer to the initial right subtree as the new starting root pointer, and let the original root be the leftchild of the new root. Finally, designate the pointer to the saved leftchild of the original rightchild as a rightchild of the new root's leftchild (which is the original root).

The drawbacks of many of the efficient search tree techniques motivated the development of the splay tree. Because binary search trees maintain sorted sets, the question arose as to whether the speed of the search process could be improved if certain items had a higher request frequency than others. In an attempt to improve performance, Allen, Munro, and Bitner proposed two self-adjusting techniques on search trees during the late 1970s. The gist of the first scheme is a single rotation of the item accessed towards the root. The second scheme involves multiple rotations of the accessed item all the way to the root. The techniques are analogous to the variations of the transpose methods for singly linked lists. Neither heuristic is efficient in the amortized sense, since long access sequences exist where the time per access is linear. It is thus clear that the search paths to frequently accessed items need to be as short as possible. Tarjan and Sleator's proposed self-adjusting heuristic halves the depth of each node on the path to an accessed item when the item is finally moved to the root. A splay tree is a binary search tree that employs this heuristic.

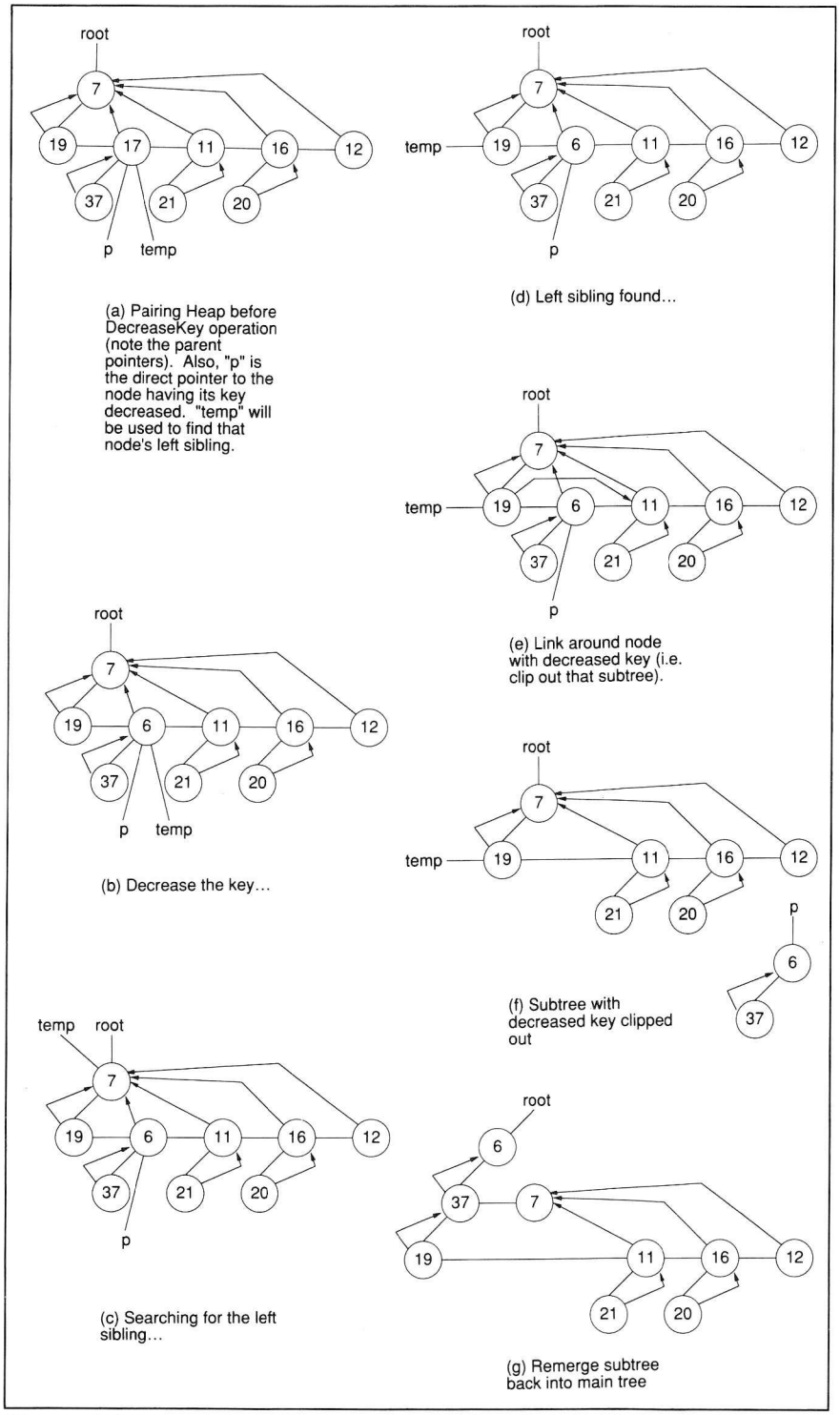


Figure 5: DecreaseKey *process on a pairing heap*

Splay Trees

The proposed self-adjusting heuristic has two versions. The "bottom-up splay" is appropriate if you already have direct access to the node that is to be moved to the root. Heuristic restructuring occurs during the second pass back up the search path (assuming that the first pass down the tree is performed in order to find the item). The second version of the proposed self-adjusting

(continued from page 54)
heuristic, called a "top-down splay," is an efficient variation of the process used to carry out Tarjan and Sleator's self-adjusting heuristic. This variant requires a pointer to the tree (call it *T*), that points to the current node to be examined during the search. This heuristic also requires two double pointer records, called *L* and *R* (for left and right subtrees), that point to all items less than or greater than the node at *T*. Figure 6 describes this step.

As illustrated in Figure 7, the splaying process repeatedly applies one of the appropriate cases until there are no more cases to apply. At this point, the leftchild of the remaining root is attached to the bottom right of *L*, and the rightchild is attached to the bottom left of *R*. The final step points the leftchild of the final remaining root to the subtrees kept by *L*, and points the rightchild to the subtrees kept by *R*. (Unlike the bottom-up variation, the top-down heuristic includes the splay step.) When the search/access for a requested node fails, change the last node on the search path into the root of the tree. This step makes the definition of all of the other operations very easy.

To search for a node, simply apply the searching process as described earlier. The process of insertion involves searching for the key *V* to be inserted. If *V* is less than the key at the root, make the node that contains *V* point its leftchild pointer to the leftchild of the root (which breaks the link from the root to that leftchild), and point the rightchild pointer to the root. Otherwise, the leftchild pointer of the node that contains *V* points to the root, and the rightchild pointer points to the rightchild of the root (and the link from the root to the rightchild is broken). The insertion step is completed by designating the node that contains *V* as the root.

The split operation is essentially the process of breaking the tree at the root in the manner described in the description of the insertion process. The process of deletion is just as easy. Perform the splay search from the key to be deleted. If the root does not contain the node with the key to be deleted, nothing happens. If the root does contain the node with the key to be deleted, keep pointers to the two subtrees at the root, perform a splay search for the maximum key in the left subtree (and designate the root of the left subtree as the new root), and point the rightchild pointer of the root to the right subtree. The join operation of two binary search trees (assuming that all items in Tree 1 are less than those in Tree 2) is simply the *non-no* operation of the delete algorithm just described.

The self-adjusting heuristic provides an alternative to the standard balancing/accounting worst-case asymptotic solutions used to develop efficient programs — and may, in fact, be the method of choice. Furthermore, the algorithms to maintain these self-adjusting data structures are both conceptually easy to understand and simple to implement in practice.

In which applications could these data structures be used to improve performance? Some possibilities are symbol table management applications (MTF lists, splay trees, and possibly in conjunction with hashing schemes), graph algorithms (skew and pairing heaps, particularly with respect to finding minimum spanning trees and the shortest paths in graphs), and other network optimization algorithms (such as splay trees, particularly in maximum/minimum network flow algorithms). Recent work by Jones, Bern, and de Carvalho, as well as my own work, indicates that some of the self-adjusting data structures do seem to perform better in practice than do conventional data structures.

Case 1 (Top-Down Zig):
If x is the root of T and y is the accessed item (with x=parent(y) then if y is a leftchild, then break the link from x to y and add x to the bottom left of R (else if y is a rightchild, add x to the bottom right of L) and now T points to y, the new root.

Case 2 (Top-Down Zig-Zig):
If x is the root of T and y (child(x)=y=parent(z)) and z are both leftchildren (or both rightchildren) then do a rotation to the right (or symmetrically, a rotation to the left), break the link from y to z and attach y to the bottom left of R (else bottom right of R) and now T points to z, the new root.

Case 3 (Top-Down Zig-Zag):
If x is the root of T and y is a leftchild of x and z s a rightchild of y (or y is a rightchild of x and z s a leftchild of y), break the link from x to y and attach x to the bottom left of R (else the bottom right of L), break the link from y to z and attach y to the bottom right of L (else the bottom left of R) and now T points to z, the new root.

Figure 6: *The top-down splay step*

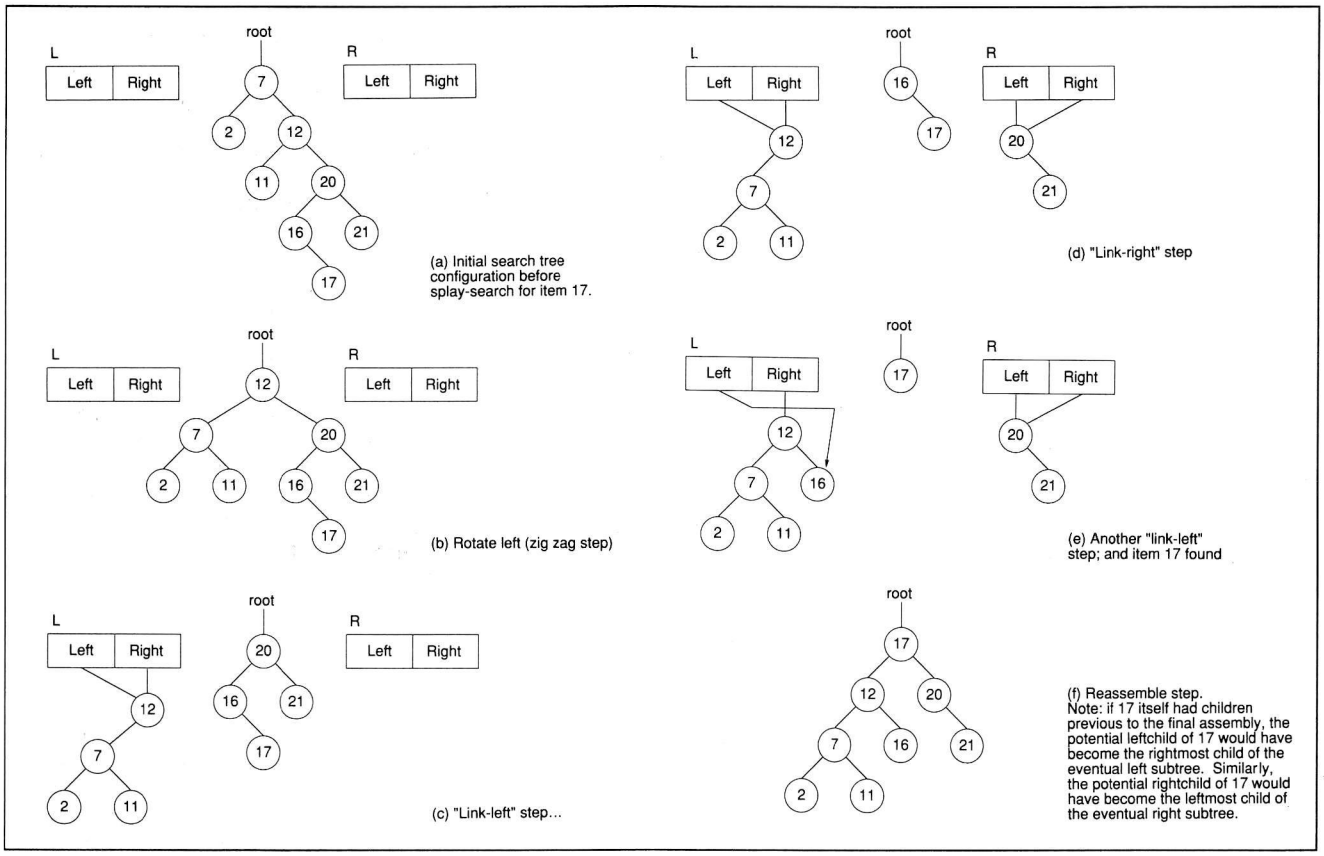


Figure 7: *Finding Item #17 in a binary search tree using the splay-move-to-root heuristic*

Bibliography

Aho, A.; Hopcroft, J.; and Ullman, J. *The Design and Analysis of Computer Algorithms.* Reading, Mass.: Addison-Wesley, 1974.

Allen, B. and Munro, I. "Self-Organizing Search Trees." *Journal of the ACM* 25 (1978).

Bentley, J.L. and McGeoch, C.C. "Amortized Analyses of Self-Organizing Sequential Search Heuristics." *Communications of the ACM* 28 (1985).

Bern, M. and de Carvalho, M. "A Greedy Heuristic for the Rectilinear Steiner Tree Problem." *Report No. UCB/CSD 87/306, Computer Science Division.* Berkeley: UC Berkeley (1987).

Bitner, J.R. "Heuristics That Dynamically Organize Data Structures." *SIAM Journal of Computing* 8 (1979).

Brown, M.R. "Implementation And Analysis Of Binomial Queue Algorithms." *SIAM Journal of Computing* 7 (1978).

Dietz, P. and Sleator, D. "Two Algorithms for Maintaining Order in a List." *Proceedings of the 19th ACM Symposium on Theory of Computing* (1987).

Fredman, M.L. and Tarjan, R.E. "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms." *Proceedings of the 25th Annual IEEE Foundation of Computer Science* (1984).

Fredman, M.L. et al. "The Pairing Heap: A New Form of Self-Adjusting Heap." *Algorithmica* 1 (1986).

Jones, D.W. "An Empirical Comparison of Priority Queue and Event Set Implementations." *Communications of the ACM* 29 (1986).

Knuth, D.E. *The Art Of Computer Programming, Vol. 3: Searching And Sorting, 2nd ed.* Reading, Mass.: Addison-Wesley, 1973.

Liao, A.M. "Three Priority Queue Applications Revisited." *Submitted to Algorithmica* (1988).

Sedgewick, R. *Algorithms.* Reading, Mass.: Addison-Wesley, 1983.

Sleator, D.D. and Tarjan, R.E. "Self-Adjusting Binary Trees." *Proceedings of the 15th ACM Symposium on Theory of Computing* (1983).

Sleator, D.D. and Tarjan, R.E. "Self-Adjusting Binary Search Trees." *Journal of the ACM* 32 (1985).

Sleator, D.D., and Tarjan, R.E. "Self-Adjusting Heaps." *SIAM Journal of Computing* 15 (1986).

Tarjan, R.E. "Data Structures And Network Algorithms." *CBMS Regional Conference Series In Applied Mathematics 44.* Philadephia: SIAM (1983).

Tarjan, R.E. "Amortized Computational Complexity." *SIAM Journal of Algebraic Discreet Methods* 6 (1985).

Vuillemin, J. "A Data Structure For Manipulating Priority Queues." *Communications of the ACM* 21 (1978).

Wirth, N. *Algorithms + Data Structures = Programs.* Englewood Cliffs, New Jersey: Prentice Hall, 1976.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 105.)

Vote for your favorite feature/article.
Circle Reader Service **No. 4.**

Multiplexing Error Codes

Improve error diagnosis

William J. McMahon

Diagnosing unexpected errors can be one of the most frustrating and troublesome aspects of software development. Even the most well-designed software can have minor flaws that have catastrophic consequences. The actual symptoms often give no clue as to the nature of the defect. Worse, they may even be difficult to reproduce, particularly if the bug is reported from the field.

A programmer can often spend hours, if not days, iterating through many cycles of program modification and testing to track down even a small defect. Fortunately there are many tools and techniques to help. Interactive debuggers, code interpreters, built-in debug code, and robust error handling within the program itself are all useful, but each has its limitations.

All but the last of the techniques just mentioned require that the program be run again to duplicate the error in question. But this is not always convenient or possible. In such cases, how the program deals with an unexpected error the first time it occurs becomes extremely important. If the error is reported in such a way that the programmer can close in on it with that information alone, much time can be saved. The major problem with unexpected errors is, of course, that they are unexpected and therefore impossible to handle specifically. They require a systematic approach.

For systematic error handling to be effective, it has to be used widely and consistently. As a practical matter, this means that error handling cannot require much, if any, extra work from the programmer.

This leaves us with two apparently conflicting goals: Providing enough information to easily diagnose unexpected errors wherever they occur, while adding little or no extra work to the original programming task. The following is a description of a scheme I have used to do just that.

```
unsigned mid_level(char *parm)
{
     unsigned err, low_level();
     if (parm == NULL)
          return (1);
     ...
     err = low_level(i, j);
     if (err)
          return (ERR_COMBINE(err, 3));
     ...
     return (0);
}

unsigned low_level(int x, int y)
{
     if (x > 0)
          return (1);
     if (x > y)
          return (2);
     ...
     return (0)
}
```

Example 1: *Combining codes and returning to the next level*

Overview

The error handling system presented here hinges on function communication. Functions that use this scheme will return an error code. A return value of zero is used to indicate success, while a non-zero return indicates some sort of failure. Exception handling logic can then be processed whenever such a function returns a non-zero value. In most cases the exception processing amounts to returning an indication of failure to the calling function. Because most low-level functions do not know the context in which they were called, they cannot deal with the error directly.

The failure returns back through several levels of functions until it is finally dealt with in some way. If it is an unanticipated error, the program will probably abort with some sort of error message. To be able to trace the root cause of the error, we need to be able to identify its source and preserve the logic path to it.

To do that this scheme associates each possible error condition within a function with a numeric code. At the lowest level that numeric value is simply returned. At each subsequent return the return value is combined with another numeric code. This uniquely identifies each return, preserving the path to the original failure. Because the path is preserved, the numeric code needs to be unique only within each function.

William is a senior programmer for Digital Products Inc. and can be reached at 108 Water Street, Watertown, MA 02172

(continued from page 60)

Consider the following example: The original error causes the return of an error code. This code uniquely identifies the location of the error within that function. After testing the return value of the function, the calling function also generates an error code that uniquely identifies the location of that call within that function.

The two codes are combined and returned to the next level. This process is repeated at each level until the error is either handled or the program is aborted. The code fragment in Example 1 shows how this might work. Note that the function *mid_level* can return both an individual error code (if *parm* is NULL) or a combined error code (if *low_level* returns an error).

The fact that the individual error numbers are hard coded might seem to violate good programming practice, but because they must be unique within each function, they should be hard coded.

Combining Codes

The actual combining of error codes is done with the macro *ERR_COMBINE*, which is the key to this scheme. Combining error codes must be done in such a way that they can be later separated and decoded.

Consider the simple scheme where the *ERR_COMBINE* macro is defined as in Example 2. Multiplying the original code by *ERR_BUMPER* before adding the new number, shifts the original left so that its value is not lost when the new code is added.

A value of 10 for *ERR_BUMPER* is convenient because the error number can be visually decoded, but when doing so it needs to be interpreted from right to left. Each digit in the decimal integer display represents the error number at each function level. The rightmost digit represents the highest-function level.

```
#define ERR_BUMPER  10
#define ERR_COMBINE(orig, to_add)    ((orig *
                              ERR_BUMPER) + to_add)
```

Example 2: *Defining a simple* ERR_COMBINE *macro*

```
void err_print(FILE *stream, unsigned err_code)
{
     do
     {
          fprintf(stream, "%d", (err_code %
                                 ERR_BUMPER));
     }
     while ((err_code /= ERR_BUMPER) > 0);
}
```

Example 3: *This function will decode the combined error code and display the individual codes*

```
main()
{
     unsigned err, function();
     ...
     err = function();
     if (err)
          abort(ERR_COMBINE(err, 3));
     ...
}
void abort(unsigned err_code);
{
     fprintf(stderr, "\n ERROR:");
     err_printf(stderr, err_code);
     exit (err_code);
}
```

Example 4: *Code produced once function failure has been located*

Decoding Error Numbers

Visual decoding is not necessary. The function in Example 3 will decode the combined error code for any value of *ERR_BUMPER* and display the individual codes so that they can be interpreted from left to right.

Consider the output *ERROR: 34152*. While this error message is far too cryptic to understand by itself, a programmer with access to the source code can pinpoint the root cause of the error quickly. The process is simple, starting with the *main* function, locate the function failure that produces code 3. See Example 4. Next, move to that function and locate the function failure in that function that produces code 4. Repeat this process at each function level until the last one is reached.

It is a good idea to check the syntax of each function call at every step. The defect is not always with the lowest-level function. I have found that when this process is complete, the bug is often obvious.

Some Improvements

The *ERR_COMBINE* macro in the previous example makes two important assumptions. First, the individual error codes are always less than the value of *ERR_BUMPER*. Second, the combining of the error codes does not overflow the data type used for the error return (*unsigned int*).

Because the individual error numbers are hard coded, the first assumption is fairly easy to control. You will find that most functions will need only a few error numbers. If your function requires much more than a half dozen, it is probably too large and should be split up into two or more smaller routines anyway. In the rare case when extra codes are needed, two codes can be combined as follows:

```
error = ERR_COMBINE
        (ERR_COMBINE(error, 9), 1);
```

The second assumption, that the error code will not overflow, is much more dangerous. In a system of any size functions will be nested at many levels. An unsigned (2 byte) *int* and an error

(continued from page 62)
bumper of 10 allows for only four or five levels of nesting before the error code will overflow. We can increase the maximum levels by using a *long* rather than an integer, as well as by reducing the value of the error bumper. However, an *int* is preferable to *long*, from a coding efficiency standpoint, because an *int* will usually be the "natural" size for the CPU (K & R p. 34). Efficiency is an important consideration because this error code will be returned and tested in many places.

We cannot guarantee that the error code will never overflow. But as Example 5 shows, a function rather than a macro lets you develop a more sophisticated scheme to save error codes that would be lost in an overflow. We can then provide support for very large systems while making no special requirements on the capacity of the error code or the size of the error bumper.

The *err_combine* function tests for potential overflow. When required, the original multiplexed code will be saved in another location and then reset, at which point normal processing will continue.

The saved code is stored in a stack implemented as an array that can be made as large as required by the program size. An additional function (*err_pop*) is required to pop any overflow portions of the multiplexed error code off the stack. The previous *err_print* function can be changed to display the entire error code, as shown in Example 6.

The source code in Listing One, page 108 combines all the aforementioned concepts into a single module of utility functions, ready for use in any new programming project.

The *err_combine* function in that module has an additional feature worth some note. It tests the original error code value, and if it is a special "pass through" value, the error codes are not combined. Only the original pass through value is returned. In this way exceptions that are expected (such as user abort) can be easily handled using the same exception processing logic. The rather cryptic abort function used in a previous example can now be expanded as shown in Example 7.

Where the case values of this switch statement are some predefined pass through values, the rule is: Any value that is an even multiple of *ERR_BUMPER* is not modified, the original value will be passed through. This rule works well because of the way that the combine algorithm works. Multiples of *ERR_BUMPER* will not be generated as error codes because individual error codes are non-zero by definition.

Summary

As mentioned earlier, the power of this error handling scheme lies in the de-

```
unsigned err_combine(unsigned original, unsigned to_add)
{
     if (original > UINT_MAX / ERR_BUMPER)
     {    /* UINT_MAX is in limits.h */
          err_push(original);
          original = 0;
     }
     return (original * ERR_BUMPER + to_add);
}
#define MAX_OVERFLOWS    10
static unsigned err_stack[MAX_OVERFLOWS];
static unsigned err_stack_top = 0;

unsigned err_pop()
{
     if (err_stack_top <= 0)
          return (0);

     --err_stack_top;

     return (err_stack[err_stack_top]);
}

void err_push(unsigned err_code)
{
    if (err_stack_top < MAX_OVERFLOWS)
    {
          err_stack[err_stack_top] = err;
          ++err_stack_top;
     }
}
```

Example 5: *Using a function instead of a macro*

```
void err_print(FILE *stream, unsigned in err_code)
{
      while (err_code)
      {
          do
          {
               fprintf(stream, "%d", (err % ERR_BUMPER));
          }
          while ((err_code /= ERR_BUMPER) > 0);
          err_code = err_pop();
     }
}
```

Example 6: *Changing* ERR_PRINT *to display the entire error code*

```
void abort(err)
{
     switch (err)
     {
     case DISK_SPACE_ERROR:
          printf("\n Not enough disk space to run program");
          break;
     case MEMORY_ERROR:
          printf("\n Not enough memory to run program.");
          break;
     case USER_ABORT:
          printf(stderr, "\n Program aborted by user.");
          break;
     default:
          printf("\n Unexpected error: %d ", err);
          printf("\n Please record this error number,");
          printf("\n and call technical support at");
          printf("\n 1-800-555-1234.");
     break;
     }
     exit(err);
}
```

Example 7: *Expanding the abort function*

tection of unexpected errors through its systematic use. It can detect errors only where it is used. To encourage its wide use, it has been designed to minimize the work required to implement it. Any time an error condition is returned by a function, it can simply be kicked upstairs without concern for losing calling context information.

Because a unique identifier is added each step of the way, valuable trace information can be provided. This information can often be critical in discovering the nature of a program defect. And by providing the information with the first occurrence of the bug, we can significantly reduce the diagnosis time, particularly for bugs with symptoms that are difficult to reproduce.

Because the trace information is preserved, the individual error codes need to be unique to the function only. This is very convenient in large systems where managing many error numbers would become quite cumbersome.

You may have guessed that this error identification scheme is most helpful in the early stages of development, and it is. But it is also helpful late in a program's life cycle. It's especially good for catching those "once in a blue moon" bugs. Usually all you need is the error code and the correct version of the source code.

You will find that this system will not always lead you to the program defect by itself, and you may have to resort to other debugging techniques, but it will provide you with a valuable head start.

Notes

The C Programming Language, by Brian Kernighan and Dennis Ritchie. Prentice Hall, Englewood Cliffs, New Jersey, 1978.

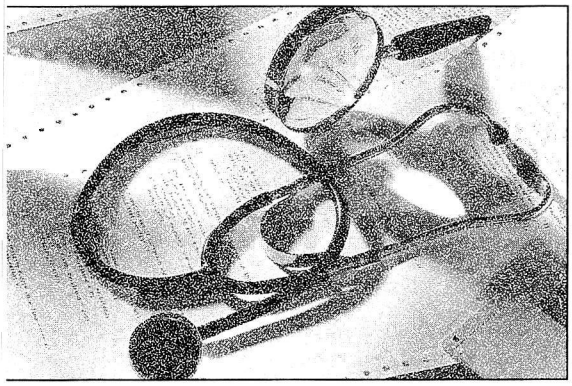
Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listing begins on page 108.)

Vote for your favorite feature/article.
Circle Reader Service **No. 5.**



C_talk/Views

Recently ported to Microsoft Windows, this unique language adds a powerful class library for dealing with graphical user interfaces

Noel Bergman

With object-oriented programming emerging as the major new programming paradigm for the 1990s, and C being the darling language of professional software developers in the PC and minicomputer arenas, what could be more natural than object-oriented extensions to C? C_talk is one such object-oriented extension to C and it is very much in the mold pioneered by Brad Cox's Objective-C language. While there are a number of such derivative languages, they all nonetheless have similar differences from the other great object-oriented C dialect, AT&T's C++.

C_talk is extended from C with two basic constructs: classes and messages. C_talk messages take the form:

```
[< lvalue > =] @< receiver >
                    < message >@
```

The optional assignment is an enhancement from earlier versions of C_talk, and allows a method to return a value. (Remember that a message is what you send to an object, and a method is the actual code that handles a message.)

Noel Bergman is the president of Development Technologies, Inc., a software development consulting firm with special interests in OS/2, Windows, and OOP. He can be reached on CompuServe (CIS ID: 76704,34) where he is a volunteer Sysop on Microsoft's CompuServe forum. Development Technologies can be reached at 8329 High School Road, Elkins Park, PA 19117 or 215-386-9599.

C_talk uses a more Smalltalk-like syntax with named parameters, unlike the C++ method/message syntax. (For more information on the C_talk syntax, see "A Class-ier C" by Ernie Tello, *DDJ* December 1988, page 74.) As with Smalltalk, C_talk's class hierarchy has a single class object, from which all other classes are derived. There is no multiple inheritance, though the true polymorphism in C_talk cuts down on the need for multiple inheritance, unlike C++.

If a language has true polymorphism, it is possible to have an arbitrary collection of objects, and to send a message to each member object without knowing its type. This is not possible in C++. Although virtual functions do provide limited polymorphism, each object in a collection must be descended from a common ancestor who declared the virtual function. One way of getting around this in C++ is to use multiple inheritance, but a detailed explanation of how is beyond the scope of this article.

C++ supports static, or early binding; this means that the function to be called at run time is known at compile time. In the case of a virtual function, the index in an array of pointers to functions is known. True polymorphism includes dynamic, or late binding, in which the method is looked up for the specific object and executed. But there is a run-time penalty for late binding. Objective-C now offers the option of mixed early and late binding. C_talk only offers late binding in the language, but a new tool, C_talk/Views, performs some early binding at compile time.

The C_talk/Views Package

C_talk/Views is a package made up from a number of components. The C_talk Browser is a specialized editor for working with object classes and methods; the C_talk preprocessor converts C_talk into C; class libraries provide a base set of fundamental object-oriented classes; and a special class library is provided for working with graphical user interfaces. C_talk also includes a *Make* facility and a Streamliner. The latter is a specialized tool that attempts to determine when early binding can be used and optimizes the code accordingly.

Most of these tools were covered in Ernie Tello's earlier C_talk review, so I'll focus on what is new in the C_talk/Views package: Windows Browser, Streamliner, Interface Generator, and Views.

The C_talk Browser

The C_talk Browser is the tool you'll use for most, if not all, of your C_talk editing. It is possible to use a normal text editor, but the C_talk-to-C preprocessor needs certain specially formatted information, so it is likely that you'll use a standard text editor only to edit some methods on occasion.

The C_talk Browser is functionally the same as the earlier Browser, which was based on text windows. In fact, a text-based Browser is included in the same package with the Microsoft Windows Browser. Figures 1 and 2 show the text-windows-based browser and the Microsoft Windows Browser, respectively. The differences are in the manipulation of menus and windows.

(continued from page 66)

Streamlining: Early Binding for Speed

The Streamliner, CTSW, is used when preparing a product for final testing. It analyzes your application, strips out classes and methods that the analysis reports cannot possibly use, and performs static binding of messages. The result is code that may be half the size of non-streamlined code, and that typically has two-thirds of the messages statically bound.

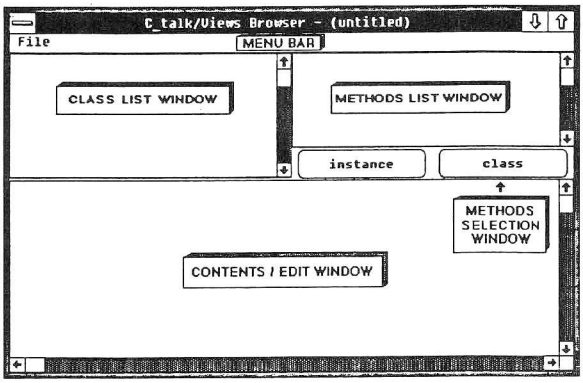


Figure 1: *Text windows based browser*

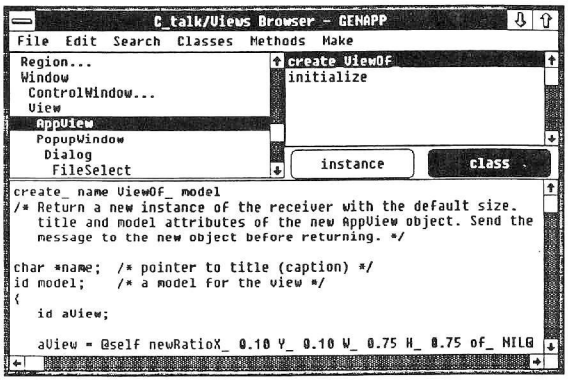


Figure 2: *The Microsoft Windows Browser*

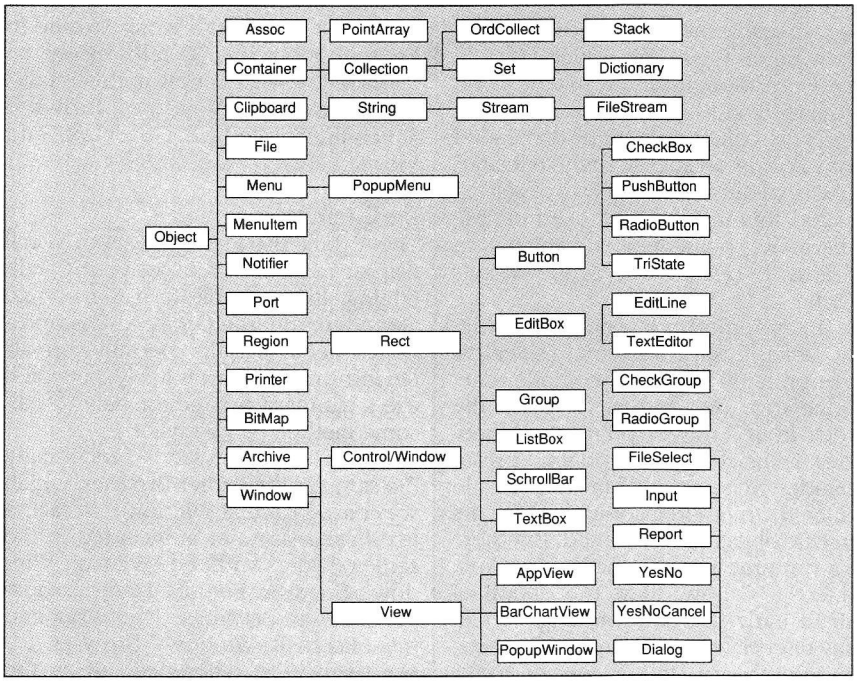


Figure 3: *A subset of the C_talk/Views classes*

The Streamliner does have some quirks, but these are minor and are actually documented in the manual.

The Interface Generator

C_talk/Views includes a CTIG utility that translates the .DLG output from the Microsoft Windows Dialog Editor into a C_talk/Views class. Basically, it writes the class method *CreateOf_* and instance method *initialize*. You edit the new class, adding instance variables and methods to flesh it out into a complete dialog subclass that is ready for use in applications. In essence, CTIG takes a description of the visual portion of a dialog and codes it for you, leaving you to code only the dialog's function.

C_talk does not allow you to build DLLs from C_talk code, although you can use DLLs written in C or MASM

Actually, CTIG takes care of one of the Windows programmer's least favorite dialog characteristics — how the code looks on-screen. CTIG generates code that will look correct on the display, regardless of the display's aspect ratio and resolution.

Currently, CTIG is limited to generating subclasses of Views' *DIALOG* class, which are framed dialogs. But those dialogs are the majority of dialogs used today.

The New Classes

OK, enough with the tools. These are just delivery vehicles for the true power of OOP, the object classes themselves. C_talk comes with a full complement of foundation object classes in the Smalltalk model. To this has been added Views, a rich set of classes for working with graphical user interfaces. Most of these classes are shown in Figure 3.

The earlier C_talk article discussed the foundation classes, so I will again focus on what is new in C_talk/Views.

Additional Foundation Classes

In addition to the new Views classes, there is a new foundation class in the C_talk/Views package that provides support for persistent objects. [*Editor's note*: See "Persistent Objects" by Charles-A.

(continued from page 68)
Rovira, *Dr. Dobb's Macintosh Journal*, Fall 1989.] A future release, which should be available by the time this article reaches print, is expected to add support for communications, timers, and debugging.

Persistence is implemented through an Archiver object class. Archiver objects take care of most of the details involved in storing and loading archives. Objects that are to be kept in the archives, however, must implement the methods *putTo_* and *getFrom_*, which are responsible for storing and loading the subclass' instance variables. This form of persistence is suitable for many applications, but does not directly support random access to objects.

Views: Classes for Windowing Systems

Previous versions of C_talk also included windowing classes, albeit text-based. The new Views classes should not be confused with those earlier windowing packages, although there are some similarities.

Model/View/Controller

The paradigm chosen to be implemented in the Views class library is a modification of the classic Smalltalk Model/View/Controller (MVC) metaphor. This framework formalizes program structure and places classes into one of three categories:

Models — Virtualize a concept without concern for its representation. For example, the hierarchy of classes in a program, or a network of workstations on a LAN.
Views — Provide a visual representation of a model. One such representation is the list of classes in the C_talk browser. Another representation of the same information would be the directed graph in Figure 3.
Controllers — Provide the interface between the other classes and input devices. The interactions between models, views, and controllers are more involved in the Smalltalk MVC model. In the Views adaptation of MVC, the notifier is the only kind of controller. This simplification works well with the Windows/PM application model, because it is analogous to the message queue. In fact, the OS/2 version of Views will probably use one notifier for each message queue thread.

The Views implementation of the MVC paradigm differs from the classic MVC paradigm (see *The Journal of Object-Oriented Programming*, August/September 1988) in a number of ways. In the latter, Controllers talk directly to both views and models. Models can

be active when talking to controllers and views. The Views paradigm has a single controller that talks only to views, and the models are passive (only responding to messages from view objects).

The Views controller is essentially a virtualization of the event system. Therefore, it does not talk directly to models. The events are processed by a view which gives meaning to the events in that view's context.

The Views paradigm is simpler, but does have a few drawbacks. You can have, and often want, multiple views to the same model. The problem is that all the views that use the model have to know about each other, in order to let each other know when a change is made. Future versions of Views may change models, so that a model keeps track of all views using it and notifies them when there is a change in the model.

Using MVC

It takes a little extra development time to use MVC. You have to decide upon formal models for the concepts your program works with. Then you will create views that use the formal interface to the model. Windows/PM programmers tend to intertwine windows (views) and the concepts they represent (models). Even the developers of C_talk/Views have done this. Only a couple of the sample programs use the MVC model, and even then use it somewhat loosely. All of the rest handle model issues as part of the view, making it more difficult to provide alternate views to the same information, change the model without changing the view, and vice versa.

A C_talk/Views Example

C_talk/Views comes with a range of sample programs, from a simple doodle application to a couple of simulations. One of those programs, Sketch (an object-oriented sketch pad), has been a staple of mine at lectures, and in fact has undergone a number of changes either at my hand or at my request. It has never used MVC, however, or the Archiver class. Until now, that is!

Sketch is made up of a few key classes (not counting dialogs or normal foundation classes, such as collections, used in most OO programs): *SketchApp* (the MVC model), *SketchView* (the MVC View), *Notifier* (the MVC controller), and a small *GraphObj* family.

SketchApp is the model, and is where I've put the *main* method (see Listing One, page 109), that current versions of C_talk/Views use as a dummy to hold the *main( )* function. It is important to remember that *main( )* is not a C_talk method — there is no self to refer to during its execution. Accordingly, *main* creates a new instance of *SketchApp*, then uses *SketchView*'s *create_ViewOf_ class* method to create a new *SketchView* with the new *SketchApp* as the model.

The paradigm chosen to be implemented in the Views class library is a modification of the classic Smalltalk Model/View/Controller metaphor

In the current implementation, *SketchApp* models the behavior of the image memory for the sketch pad. New objects are drawn on the *SketchView*, which sends an *add_* message to the model. The model records all of this information, is queried when the view wants to repaint the entire image, and handles the archiving of the image.

SketchView handles the user interface for the sketch program. It is rather simple in structure, primarily because of Views' menu handling and C_talk's polymorphism. The initialize method (see Listing Two, page 110) tells the model to clear (initializing it to a known, empty state), create a *Port* object (the Views class, which virtualizes the drawing surface), and set up a menu of functions — not in that order. A Views menu is made up of 3-tuples [name, accelerator, method]. When a menu item is selected, the associated method is activated for the object associated with the menu (in our case, the *SketchView* object).

The three graphic "shapes" in the current Sketch program are lines, boxes, and free-hand drawing, each of which is selected from a menu item. The method associated with each menu item is almost identical and is simple. In general, the method unchecks any checked items on the menu (in order to clear the current shape indicator), sets the *drawClass* instance variable to the class for subsequent shapes, and checks the menu item for that shape. (See Example 1.) When a mouse-down

event occurs, the view will create a new instance from *drawClass* and send mouse messages to it; drawing is virtualized by the classes installed in *drawClass*.

SketchView, which is the Views object in our MVC model, receives events from the controller. The Views notifier gives us control over a number of things, one of which is the ability to capture the mouse. *SketchView*'s *mouseDnX_Y_* method is sent mouse-down events from the notifier. If there is a current drawing class, we will turn mouse capture on, and enable tracking of mouse movement (otherwise Views doesn't waste time dispatching those events). Then a new drawing object is created, added to the model, associated with the drawing port for our window, and given the mouse-down event for class-specific handling. Similarly, mouse movement and mouse button up events will first be processed by the *SketchView*, then passed to the drawing object. *SketchView* handles the interaction with the controller, but leaves all of the drawing to the drawing object. In order to add a new shape, all you have to do is create a class for it and add it to the menu.

All of these drawing classes are grouped under the *GraphObj* class. Each subclass specializes the *mouseDnX_Y_*, *drawOn_*, *mouseMvX_Y_Bt_*, *mouseUpX_ Y_*, and *setPort_* methods, which are invoked from *SketchView*. Also added are the *getFrom_* and *putTo_* methods for the Archiver class.

Archiving is rather straightforward, once you get the hang of it. Each class must implement *getFrom_* and *putTo_* methods that are only to be called from the archiver — never directly! These methods take an instance of Archiver as the parameter and are usually responsible for restoring or saving the classes instances variables. (If there are no new instance variables, the methods inherited from the parent class are usually sufficient.) A typical method first sends the message to super to affect inherited instance variables, then handles those specific to its subclass.

SketchView has been enhanced with menu items and methods (*open*, *save*, and *saveAs_*) for interacting with the user and sending the user's request to the model. *SketchApp* implements *loadFrom_* and *save* methods (*saveAs* just sets the file name and invokes *save*), which it uses to archive the collection of *GraphObj* instances.

Archivers are easy to use. There are two primary methods, *getObject* and *putObject_*, which are sent to an instance of *Archiver* to get the next object from the archive or to put another one in. *SketchApp* uses them to save and restore the objects instance variable. Some of the *GraphObj* classes have objects as instance variables; those, too, are sent to the *Archiver* with *putObject_* and retrieved with *getObject*. Other instance variables are integers, longs, or other base types. There are *Archiver* methods for these, too, such as *putInt_* and *getInt*.

This covers the high points in the implementation of the new Sketch program. The key changes in this new version are the move towards MVC, and the use of the *Archiver* class.

One major change in the Sketch program was made prior to the first public release, but it bears mentioning here. The first version I saw, prior to adopting the program for my own uses, didn't use *GraphObj* to virtualize the drawing, and didn't have a persistent image — if you moved the window, the image was erased. Instead, the menu items set an instance variable ivar to indicate which shape was being drawn, and SketchView's mouse handling methods used *if* statements to check the ivar and do the appropriate things. The change to using *GraphObj* and an ivar that pointed to the current class not only provided better polymorphism, but led the way to a persistent image, MVC, and archived graphics. Polymorphism and virtualization are key concepts and cannot be too highly emphasized.

On the Downside

C_talk/Views is certainly not a perfect tool. For one thing, it is not written in C++. That's both good and bad. C++ has a lot of inertia behind it as it rolls towards being a standard, but it has limitations, some of which I discussed at the beginning of this article.

C_talk does not allow you to build DLLs from C_talk code, although you can use DLLs written in C or MASM. Views neither implements nor provides help for implementing MDI. These are not expected to change in the near future.

```
*** Generic "shape" method for SketchView ***
<shape>_ aMenuItem
id  aMenuItem;  /* id of selected menu item */
{
  @self->fMenu uncheckAll@;
  self->drawClass = <ShapeClass>;
  @aMenuItem check@;
}
```

Example 1: *A generic* shape *method*

(continued from page 72)

Another problem, which is expected to be fixed in the version available before this article is published, is that all C_talk source files currently have to be located in the same directory. The new version of C_talk will allow you to keep standard classes in one directory and application-specific classes in another.

Finally, there are some annoying little quirks in the browser. For example, there is a lack of keyboard accelerators for some of the menu items. Also, the .DEF file generated by the browser doesn't have the EXETYPE WINDOWS entry required by current versions of the Linker. These are expected to be addressed in the next release.

On the Upside

C_talk/Views permits rapid development of commercial quality Windows programs. This is partially because of the nature of the language, but is mostly due to the robust class library. Even such things as printing are standard in the C_talk/Views package. The Views classes simplify many Windows-related tasks, particularly menu handling and the repositioning of child windows when resizing their parent.

What's Next?

Part of the promise of OOP is portability, not just ease of development. Future versions of C_talk/Views are targeted for other platforms besides Microsoft Windows. Presentation Manager, Macintosh, and X Windows are all being considered. Also, Views will be ported to other languages, such as C++ and Actor.

Personally, I like the possibilities of starting work in Actor/Views, working in a completely interactive environment, then porting the code to C_talk/Views to take advantage of improved memory and CPU utilization. The languages provide similar constructs; the major reason not to do it today is that you would have to change your windowing model, but an Actor version of Views would eliminate that problem. Smalltalk/Views or Object/1 Views would be similarly interesting.

All in all, programmers who harness the power of the C_talk/Views package should reap today's all-important harvest, with even better yields in the future.

Products Mentioned

C_talk/Views
CNS, Inc.
Software Products Dept.
7090 Shady Oak Road
Eden Prairie, MN 55344
Price: $450.00
Requirements: MS-DOS 2.0, MS Windows
Microsoft C 5.0
MS Windows Software Development Kit 2.0.
Suggested: 80286-based computer, minimum 1-Mbyte RAM

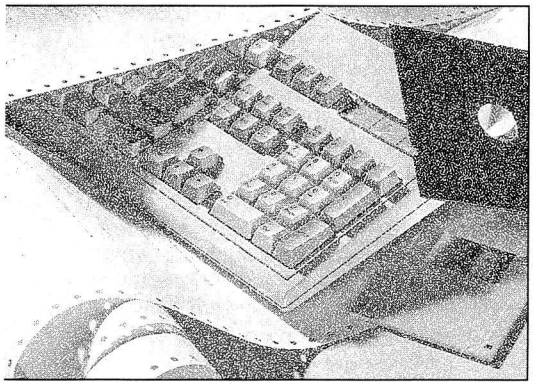
Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 109.)

Vote for your favorite feature/article.
Circle Reader Service **No. 6.**



Stalking GP Faults: Part II

The hunt continues

Andrew Schulman

Conventional wisdom says that when a protected-mode program commits a general-protection (GP fault) violation, it is "evidence that the program's logic is incorrect, and therefore it cannot be expected to fix itself or trusted to notify the user of its ill health." This is a quote from OS/2's principal architect, Gordon Letwin. In Part I of this article I tried to show that this statement is false. For a large class of programs, which execute some form of "user code" or script (a programmable data base or a Basic interpreter with PEEK and POKE commands, for example), GP faults are caused by the user code, not the program itself. An extra module should be written to catch GP faults when such programs are ported to protected mode (either through a DOS extender or OS/2).

In Part I of this article, I showed how to catch GP faults under a 286-based DOS extender such as Rational Systems's DOS/16M. I used a rather silly "GP fault interpreter," a program that allows the user to commit GP faults. In this article, I will show how to do the same thing using 32-bit C compilers and 386 | DOS-Extender. Then I will show how GP faults can be caught under the OS/2 operating system on 16-bit machines. It is often said that GP faults can't be caught under OS/2, but they can (and must!).

Andrew is a software engineer in Cambridge, Mass., where he is writing a network CD-ROM server. Andrew can be reached at 32 Andrew St., Cambridge, MA 02139.

32-bit MS-DOS

Like a 16-bit DOS extender, Phar Lap Software's 386 | DOS-Extender runs code in protected mode, occasionally switching back to real mode to call MS-DOS or some other real-mode service. In many ways, programming for this 32-bit environment is similar to 16-bit protected-mode programming.

There are important differences, however. In a 32-bit C compiler, such as MetaWare High C v. 1.5 for MS-DOS 386 or Watcom C 7.0/386, an *int* is a 4-byte quantity. A near pointer is also a 4-byte quantity. This means the programmer almost never have to deal with far pointers. When a segment takes a 4-byte offset, a program only needs one segment for all its data, and another segment for code. Once loaded, DS and CS stay constant. In effect, this is a linear address space.

Though it is no longer needed for data and code, segmentation is still essential for sharing and to enforce protection. In a DOS extender, segmentation is also needed for communicating with real-mode services. On those occasions, when both a segment and an offset are needed, a far pointer is a 6-byte quantity (an FWORD).

The basic difference between working in 32-bit versus 16-bit mode is that there are fewer restrictions. It takes longer for an *int* to overflow, the registers (EAX, EBX, and so on) are twice as wide, there are more registers (FS, GS), and offsets can be so large that they are full-fledged addresses. All these quantitative differences add up to a major qualitative change. Some of the flavor of 386 protected-mode programming appears in a sample session with a 32-bit version of the GP fault interpreter, compiled with Watcom C 7.0/386, and running under 386 | DOS-Extender, as shown in Figure 1.

Except for the extra 386 registers, Figure 1 resembles the DOS/16M GP Fault interpreter shown in Part I. As in the

```
C:_PHARLAP>run386 gpf386
'Q' to quit, '!' to reinstall default GP fault handler
0014:000038a4 is a legal address to poke
0014:000038a0 is not a legal address to poke
$ 1234:00005678 66666666
Protection violation at 000C:00000111!
Error code 1234
<DS 0014> <ES 000C> <FS 0014>          <GS 0014>
<EDI 00003CC0> <ESI 66666666>          <FLAGS 00010297>
<EAX 00005678> <EBX 00005678>          <ECX 00001234> <EDX 00001234>
$ 000c:00000111 66666666
Protection violation at 000C:00000127!
<DS 0014> <ES 000C> <FS 000C>          <GS 0014>
<EDI 0000000C>  <ESI 66666666>         <FLAGS 00010212>
<EAX 00000019>  <EBX 00000111>         <ECX 0000000C> <EDX 0000000C>
$ 0034:000b80a0 70217021
poked 0034:000b80a0 with 70217021
```

Figure 1: *A sample session with a 32-bit version of the GP fault interpreter*

(continued from page 76)
16-bit version, trying to poke at segment 1234 failed, though this time the faulting instruction (*EIP=00000111*) was:

```
mov fs,dx
```

Trying to write into the program's code space also failed. The faulting instruction at *EIP=00000127* was:

```
mov fs:[ebx], esi
```

The last POKE command in Figure 1 succeeded, though. In 386|DOS-Extender, 0x34 is a writeable data segment that maps the entire first megabyte of memory. The entire MS-DOS address space occupies a tiny portion of one 386 protected-mode segment! Thus, *0034:000B80A0* is equivalent to the real-mode address *B800:00A0*, and points into video display memory. Poking the integer *0x70217021* into this address, after GPF386.EXP scrolls the screen, leaves two reverse video (attribute 0x70) exclamation marks (*char* 0x21) in the upper-left corner of the screen.

Listing Four, page 112, is the Watcom C version of GPF386.C. Listing Five, page 112, is the MetaWare High C version. The object file produced by the compiler is passed to Phar Lap's 386LINK. As the example shows, the resulting file, GPF386.EXP, runs under RUN386. Those who purchase a redistribution package from Phar Lap can bind RUN386 into a protected-mode application, thus producing the standalone executable file GPF386.EXE. The process is similar in DOS/16M.

386|DOS-Extender requires Set Vector to handle 32-bit offsets; placing the address of a protected-mode interrupt handler in DS:EDX, or using EBX to hold the entire address of a real-mode handler. MS-DOS INT 21 function 25 won't work here, so Phar Lap provides its own.

Phar Lap makes INT 21 function 25, the gateway for all 386|DOS-Extender system calls. For example, AX=2504 is used to set a protected-mode interrupt vector and AX=2505 is used to set a real-mode interrupt vector. Other 386|DOS-Extender system calls let the programmer call a real-mode procedure, issue a real-mode interrupt, alias a segment, change the attributes for a segment, and so on.

Phar Lap provides several different system calls for setting interrupt handlers. The most useful one for catching GP faults is the "Set Interrupt to Always Gain Control in Protected Mode" call (*AX=2506*). Because this call expects the address of a protected-mode handler in *DS:EDX*, this is one of those times that DS has to temporarily change. The function *setvect( )*, shown in Listings Four and Five, demonstrates this process.

High C provides a C interface file, msdos.cf, with declarations for *calldos( )* and for a global Registers structure. This is similar to using *intdosx( )* and *union REGS *r* when accessing DOS from a Microsoft C or Turbo C program.

Watcom C7.0/386 goes further. Watcom C is a 32-bit compiler that is highly compatible with the Microsoft C 16-bit standard. Watcom C's *dos.h #include* file contains functions such as *_dos_getvect( )* and *_dos_setvect( )*. These functions invoke Phar Lap system calls for protected-mode handlers. Such as their Microsoft C equivalents, they use far pointers, except that a far pointer in Watcom C is 48 bits. Like the standard C library MS-DOS interfaces, Watcom C7.0/386 includes the *intdosx( )* function and *union REGS ** declaration, but these manipulate the 32-bit registers expected by 386|DOS-Extender.

I didn't use these functions for the program shown in Listing Four. Instead, I used Watcom's *#pragma aux*, a high-level facility for inline machine code. It is different from other machine-code facilities, such as Turbo Pascal's dreaded *inline( )*, which returns programmers to the days before there were even assemblers. One way *#pragma aux* can be used is to specify how a function takes its parameters, and how it returns a value to its caller. John Dlugosz discusses this facility in his September 1989 *DDJ* review of Watcom C7.0. I will be discussing it some more in a forthcoming *DDJ* review of Watcom's 7.0/386 compiler. Listing Four gives an extended example.

For the GP Fault interpreter, a segment and an offset must be merged into a far pointer. Because C does not have any built-in FWORD data types, it is tricky to write a 386 version of the *MK_FP( )* macro. In High C, I set the segment and offset portions of the far pointer separately, using the *FP_SEG( )* and *FP_OFF( )* macros in GPFAULT.H (see Listing Two in Part I). Watcom C7.0 386 uses the *#pragma aux* facility to provide a 48-bit *MD_FP( )* that is analogous to the 32-bit *MK_FP( )* macro provided in Turbo C.

There is one more area where Watcom C made it a lot easier to port the GP Fault interpreter to the 386 than did High C. While both Watcom C and High C have facilities to write interrupt handlers, Watcom C pushes FS and GS onto the stack of a 386 interrupt handler.

Catching OS/2 Faults with DosPTrace

Because it runs in protected mode,

(continued from page 78)
OS/2 has some resemblances to 16-bit DOS extenders. Except for system calls, code that runs in a DOS extender such as DOS/16M also runs under OS/2. Many features of OS/2 have nothing to do with Microsoft, IBM, or OS/2 per se. They are features of Intel's 286. OS/2 is very much a 286 operating system, even when running on a 386 machine. Programming for OS/2 has more in common with programming for DOS/16M than does programming for 386|DOS-Extender.

There are, however, crucial differences between OS/2 and a DOS extender. A DOS extender is only a front-end to MS-DOS, whereas OS/2 is a full-fledged operating system. Because they made a clean break from DOS, rather than extending it, OS/2's designers were able to junk the INT 21 interface. OS/2 system calls are invoked by putting arguments on the stack, and doing a far CALL. No more stuffing registers.

With DOS extenders, INT 0D handlers are installed with INT 21, function 25. In OS/2, interrupt handlers are installed by calling the *DosSetVec( )* function. There is a problem, however: *DosSetVec( )* only allows certain exceptions, and the GP fault isn't one of them. Another function, *DosSetSigHandler( )*, might be expected to work for SIGSEGV, but it doesn't. Even though Microsoft C for OS/2 includes SIGSEGV in *<signal.h>*, it doesn't do anything.

This is not an OS/2 bug or oversight, but a conscious design decision and, some feel, an important design flaw. In his book *Inside OS/2*, Gordon Letwin, Microsoft's chief architect for system software, flatly states, "Applications cannot intercept general protection fault errors. . . . The OS/2 design does allow almost any other error on the part of an application to be detected and handled by that application. For example, 'Illegal filename' is an error caused by user input, not by the application."

This overlooks applications in which illegal memory access is as easy for the user as illegal file access. This is particularly ironic because, in other respects, OS/2 invites one to write such programming-on-the-fly environments. I have heard that in the 386 version of OS/2 (OS/3?), *DosSetVect( )* will allow installation of a normal interrupt handler for INT 0D. This is just a rumor and, in any case, OS/3 won't be available for some time.

In the meantime, there must be some mechanism in OS/2 to catch GP faults. In fact, there is such a mechanism and, if you've ever used protected-mode CodeView (CVP), you've probably seen it in operation. Take a buggy application such as the one at the beginning of this article and run it under CVP. Instead of OS/2 displaying its familiar GP fault register dump, CVP displays the message "Segmentation violation." You can reassemble the faulting instruction, or move different values into the registers, and resume execution.

If CodeView can catch GP faults, why can't we? I asked this question on CompuServe about a year and a half ago, when I was porting David Betz's XLISP to OS/2. Ray Duncan and Charles Petzold supplied the answer: Anything that CVP can do, including catching GP faults, other OS/2 applications can also do. CVP is built on top of an OS/2 kernel function, *DosPTrace( )*, and this function — the CodeView engine — is available to all OS/2 programs. There are better OS/2 debuggers than CVP, but these too are undoubtedly written using *DosPTrace*.

Unix programmers will recognize that *DosPTrace* is process trace (*ptrace( )*), used to implement breakpoint debuggers such as sdb. In the "bad old days" of the $3000 OS/2 SDK, when developers asked for information on *DosPTrace*, Microsoft referred them to *ptrace( )* in the Unix manual. Even to-

day, while *DosPTrace* does appear in the OS/2 programmer's reference, the best source of information about it is an untitled Microsoft document, PTRACE.DOC, which is available on a number of bulletin boards.

A process (usually a debugger) uses *DosPTrace* to trace another process. To write a program that can catch its own GP faults, though, I will use *DosPTrace* in a control/event loop. Using a technique devised by Ray Duncan, the GP Fault interpreter will run itself under *DosPTrace*. Unfortunately, one thread of a process running *DosPTrace* cannot trace another thread in the same process, so the program is split into two processes. OS2TRACE.C appears in Listing Six, page 114.

DosPTrace takes one parameter, a pointer to a PTRACEBUF. This structure is declared in an OS/2 header file, and is available if *INCL_DOSTRACE* appears in a *#define* directive before the *#include os2.h* statement. A PTRACEBUF contains fields for all the 286 registers, fields to specify the process ID and thread ID and, most importantly, a field used to issue *DosPTrace* commands and get back *DosPTrace* event notifications. Symbolic names for these commands and events are in PTRACE.H (Listing Seven, page 115).

There are many *DosPTrace* commands, including *SINGLE_STEP, WRITE_I_SPACE* (write instruction space, that is, make code), *WRITE_D_SPACE* (write data space), and so on; the one used here, *GO*, simply runs the child process. All threads of the child process run until something "interesting" happens. At that point, *DosPTrace* returns and the caller can see what event took place. In Listing Six, naturally, I am mainly interested in *EVENT_GP_FAULT*.

There is very little performance overhead when a program is run under *DosPTrace* using the *GO* command (using *SINGLE_STEP*, the debugger would run as slowly as molasses). A GP fault, though, is a very expensive operation. In one test under OS/2, I could only commit about 200 GP faults per second. This is acceptable, because GP faults should take place infrequently.

When a user runs OS2TRACE, the program *execs* another instance of itself under *DosPTrace* and, using a command-line argument, tells this second process that it is the second OS2TRACE process and therefore should run the GP fault interpreter rather than the *DosPTrace* loop. If the user causes the interpreter to fault, *DosPTrace* returns *EVENT_GP_FAULT*, and the process running *DosPTrace* detects that the interpreter process has GP faulted.

The *DosPTrace* process must communicate this fault back to the interpreter process, so that the interpreter can resume execution at a different CS:IP. When the trace process detects a GP fault, it uses the *DosPTrace WRITE_REGISTERS* command to alter the CS:IP of the interpreter process. When the tracer next tells the interpreter to *GO*, the interpreter resumes at a new location.

Extensible systems need to react to and recover from errors after they happen, or after some underlying system has detected them. Protected mode is just such an underlying system, and we should take advantage of it

Where should the interpreter jump? In C, it is difficult to get the address of an arbitrary line of code. Because there is no equivalent to the $ location counter used in MASM and other assemblers, OS2TRACE.C uses the address of a parameterless function. This one-liner *longjmps* to the interpreter's top-level input loop. The trace process does not call this function in the interpreter process. The tracer tells the interpreter to call *(goto)* this function.

Even though the two processes share the same code, they are different processes. The tracer knows the address of this function in the interpreter's address space because of OS/2's "disjoint LDT space" — the code segment containing this function is mapped to the same slot in each process's Local Descriptor Table.

Despite this, in the program shown in Listing Six I decided to use still another *DosPTrace* command, *SEG_NUM_TO_SELECTOR*. Given a logical segment number (such as that found in a .MAP file) and a process ID, this operation returns the actual segment selector for the process. I know that the function *catch_sig_segv( )* is in segment #1. In the trace process, the function

send_sig_segv( ) first calls *selector_from_segment( )* to get the new CS for the interpreter process, and then calls *set_csip( )* to change the interpreter's registers.

Being able to catch GP faults under OS/2 is still important even for developing applications that do not run user code. Even for normal applications where a GP fault is the sign of an internal bug, OS/2's GP fault register dump is unattractive, makes little sense to most users, and can't be redirected to a file. The code in Listing Six can be modified to have the GP fault handler dump the register state to a file instead of attempting recovery. The *DosPTrace READ_I_SPACE* command can be used to disassemble the faulting instruction at CS:IP. Then, if the program ever GP faults, it could ask the user to please send you this "core dump" file.

Faulting Inside OS/2 DLLs

Because OS/2 is a far richer and more complicated environment than a DOS extender, simple peeks and pokes are not an adequate test for catching GP faults. What happens if the user of an OS/2 interpreter uses an illegal address while calling a routine in a dynamic link library (DLL), or while making a *DosXxx( )* call to the OS/2 kernel? Lugaru's Epsilon EMACS editor, OS2XLISP, UR/Forth, and the mini-interpreter I built in the November 1989 *DDJ* ("Linking While the Program Is Running"), are all OS/2 programs that let the user call DLL routines at run time, and all can fall prey to the protected-mode interpreter problem.

In the OS/2 GP fault interpreter in Listing Six, instead of poking at addresses, the user types in a number from 0 to 7. Each corresponds to a different line of bad code. Figure 2 shows a sample session, with the lines of code appended as comments.

The first two pieces of bad code hold few surprises. In the first case, the NIL pointer (*(int far *) 0L*) was loaded into ES:BX, but trying to dereference it caused a GP fault. In the second case, trying to peek at (*(int far *) -1L*) faulted earlier: The processor refused to load ES with FFFF. The error code is FFFC, not FFFF, because while one of these processor error codes looks like a segment selector, the bottom two bits are used for other purposes.

The next piece of bad code does not cause a GP fault. I wanted this code to illustrate an attempt to poke the code segment. However, I coded the example incorrectly so that instead of faulting, it successfully pokes an "x" somewhere in the data space. Strings that go first, and for the rest of this session, the word "violation" is printed out as "vxolation" This illustrates the limits of protection. The Intel processor was powerless to stop this "vxolation" of

```
C:_OS2\PTRACE>os2trace -v
$ 0          ;; x = *((int far *) 0L);
GP fault (error 0000) at 0047:01C0
AX=0030   BX=0000   CX=0000   DX=0087   SI=0087   DI=0001   BP=0CBE
DS=0087   ES=0000   IP=01C0   CS=0047   FL=2246   SP=0C5A   SS=0087
General Protection violation!
$ 1          ;; x = *((int far *) -1L);
GP fault (error FFFC) at 0047:01BE
AX=0031   BX=FFFF   CX=0000   DX=0087   SI=0087   DI=0001   BP=0CBE
DS=0087   ES=0087   IP=01BE   CS=0047   FL=2202   SP=0C5A   SS=0087
General Protection violation!
$ 2          ;; *((char *) main) = 'x';
Executed statement 2
$ 3          ;; x = VioWrtTTY(0L, 100, 0);
GP fault (error 0000) at 00D7:27F1
AX=0000   BX=02C4   CX=0051   DX=0000   SI=0087   DI=0051   BP=0C34
DS=00DF   ES=0000   IP=27F1   CS=00D7   FL=2246   SP=0C26   SS=0087
Faulted inside DLL code
General Protection vxolation!
$ 7          ;; x = DosGetInfoSeg(1L, 2L);
Thread 1 dying
Process 79 dying
C:\OS2\PTRACE>
```

Figure 2: *A sample session using the OS/2 GP fault interpreter*

```
Session Title: CVP app OS2TRACE.EXE
    The system detected a general protection
    fault (trap D) in a system call.
    CS=0047 IP=0237 S=0087 SP=0C5A
    ERRCD=0000 ERLIM=**** ERACC=**
    Arguments used in system call (high to low):
    0000 0001 0000 0002
        End the program
```

Figure 3: *An OS/2 message, different from a normal GP fault dump*

(continued from page 82)
data space.

In the next example, a bad pointer is passed to *VioWrtTTY( )*. OS2TRACE detects that the fault took place inside DLL code. DLL code uses its own data segment, but uses its caller's stack. Looking at the function *set_csip( )* in Listing Six. If, at the time of a GP fault, the interpreter process' does not equal SS, it means this small program was using someone else's DS. This means DS must be reset to a proper value before resuming execution of the interpreter. *longjmp( )* can't be relied on to restore DS, because the *jmp_buf* itself resides in the process's data segment. Before the interpreter can run again, DS must be pointed back to the correct data segment. This small program did this by using SS, but in a larger program with multiple data segments, the trace process probably would have to keep a list of valid data segments.

In the final example, bad pointers are passed to an OS/2 kernel routine, *DosGetInfoSeg( )*. A GP fault is generated, but in this case OS2TRACE is not able to catch it. This is a limitation of *DosPTrace( )*, not OS2TRACE. If this same code is run under CVP, CVP won't catch the fault either. Instead, OS/2 displays a somewhat different message than its normal GP fault dump, as shown in Figure 3. All CVP can do is display the message "Thread terminated normally (13)." The thread returns 0x0D to indicate that it has GP faulted. It's a shame that OS2TRACE can't catch this fault, but it is somewhat consoling that CVP can't either. This is a limitation of *DosPTrace*. Any OS/2 debugger (such as Logitech's MultiScope) will undoubtedly have the same limitation.

One line of bad code in OS2TRACE was not executed in the sample session. *VioWrtTTY(0L)*, in which I accidentally left off the last two arguments to *VioWrtTTY*, doesn't GP fault. Instead, it hangs OS/2! I have only found one way to make this fault inside *VioWrtTTY* without hanging the machine, and that is to single-step through the code in CVP. In that case, OS/2 detects INT 0C, the stack exception, as shown in Figure 4. CodeView prints out "Thread terminated normally (12)."

```
SYS1942: A program attempted to reference storage
outside the limits of a stack segment. The program
was ended.
TRAP 000C
```

Figure 4: *Detecting INT 0C under OS/2*

ON ERROR and ESTAE

Having figured out how to use *DosPTrace* to catch most GP faults in OS/2, it becomes clear that *DosPTrace* is a powerful part of OS/2 and could probably be used for all sorts of tricky programming. On the other hand, why is it so much more difficult to catch GP faults in OS/2 than when using a DOS extender? Part of the reason is that OS/2 is a far more ambitious undertaking than a DOS extender. A DOS extender doesn't have to worry about GP faulting inside a dynamic-link li-

Lessons from History: IBM Mainframe Error Handling

Gordon Letwin's assumption, that a GP fault is evidence that the program's logic is incorrect, is, according to Karl Finkemeyer of IBM ASD, the same mistake the early OS/360 designers made. It ignores the fact that there are situations where risking a fault condition is definitely preferable to checking each and every pointer before using it. So when PL/I came along, and with it pointers in a high-level language, OS/360 had to add a facility for the PL/I run-time environment to clean up after a user program bombed with a stray pointer. The only alternative would have been to do validity checking of every pointer before every usage, and that was intolerable performance wise.

So OS/360 first extended the existing SPIE macro (SPecify Interruption Exit) which originally was only meant for arithmetic errors (divide underflow, floating point significance check, and so on.). When SPIE became more and more unwieldy because it had to handle more and more error conditions, the non-arithmetic error conditions were taken out again and put into the new STAE (Specify Task Abnormal termination Exit) macro in MVT (Multiprogramming with a Variable number of Tasks, introduced in 1967). Abnormal termination of a task can be intercepted through the use of STAE.

The syntax ON ERROR could have originated in PL/I under MVT. The PL/I compiler under MVT translated the ON into a simple STAE exit routine. So STAE and ON ERROR probably are not just similar; they may turn out to be the same.

In IBM's MVS (Multiple Virtual Storage operating system, introduced in mid-1974, for the larger System 370 model, and their followons such as the 3090s), STAE was extended, so it became the ESTAE macro. Inside an ESTAE exit routine, you can do nearly everything you want (even restart the task) as long as you don't bump into another abnormal termination condition. So this makes it easy for a program to try dangerous things, clean up after the fact if something went wrong, and continue.

Because ESTAE is too unwieldy for high-performance system code, another mechanism was introduced there: FRRs (Functional Recovery Routines) that can be used only inside the MVS kernel, mainly because they use fixed control blocks so that the overhead of activating and deactivating them remains small. By now, every MVS routine is either associated with an FRR or is covered by its caller's FRR.

According to A.L. Scherr ("IBM Systems Journal," Vol. 12, No. 4), "An interesting footnote to this design is that now a system failure can usually be considered to be the result of two program errors: The first, in the program that started the problem; the second, in the recovery routine that could not protect the system." If a system module bombs and the FRR runs into problems and none of the more general FRRs higher up on the FRR stack can resolve the problem, only then MVS crashes. The result of this architecture is MVS's "continuous operation:" There are many installations where MVS just keeps running (even when one or more of the processors die, and are restarted after repair) until it is taken down for applying maintenance.

(continued from page 84)
brary, because DOS extenders don't provide dynamic linking.

The major reason for the difficulty, however, is that OS/2 does not provide much support for exception handling by applications. This is surprising for two reasons. First, with Microsoft's predilection for Basic, one might have expected the company to at least provide OS/2 with something such as one of Basic's most powerful features, ON ERROR (from the ON statement in PL/I). Second, and more important, if anything from IBM was going to rub off on Microsoft and OS/2, it should have been the strong emphasis on error handling, exception handling, and fault recovery found in large IBM operating systems. (See the sidebar, "Lessons from History," for a brief discussion of the ESTAE and FRR error-handling facilities in IBM's MVS.)

In Conclusion

Having spent so much time talking about interrupts, errors, faults, traps, and exceptions, by now the reader must feel that protected mode is "The Promised Land of Error" (the subtitle of a book that has nothing to do with Intel processors). Nonetheless, this discussion of catching GP faults has only scratched the surface of protected-mode interrupt handling. For example, this article never explained the difference between an exception and an interrupt. In addition to the standard Intel literature, three good books for more information on protected-mode programming are John H. Crawford and Patrick P. Gelsinger's *Programming the 80386* (Sybex, 1987), Edmund Strauss's *Inside the 80286* (Prentice-Hall, 1986), and Phillip Robinson's *Dr. Dobb's Toolbook of 80286/80386 Programming* (M&T Publishing, 1988).

Errors, exceptions, and faults are an extremely important part of programming. One author distinguished between "good" exceptions and "bad" exceptions, saying that with good exceptions, "the corrective actions you perform are an integral part of your system" and that good exceptions "are the ones you expect to occur," whereas bad exceptions indicate a program bug (Strauss, *Inside the 80286*). Using this definition, I hope I have shown that the GP fault can be a "good" exception, that many systems should expect it to occur, and that catching and recovering from it should be an integral part of many (but by no means all) systems.

Flexible, extensible systems don't need more error checking. They need error handling. The more flexible the system, the less it knows about the types it operates on, and the less up-front checking it can do. Extensible systems need to be able to react to, and recover from errors after they happen, or after some underlying system has detected them. Protected mode is such an underlying system, and we should take advantage of it.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 112.)

Vote for your favorite feature/article.
Circle Reader Service **No. 7.**

Listing One *(Text begins on page 16.)*

```
        page 78,132
        ?PLM    =       0
        .xlist
        include CMACROS.INC
        .list
;Use C calling convention

        .model  medium
        .code

cProc   memcpyifp,<PUBLIC>,<ds,si,di>
parmD   DestIfp
parmD   SourceIfp
parmW   cb
cBegin
        mov     bx,SEG_DestIfp
        mov     dx,bx                   ;pSeg is high word of return value
        mov     es,ds:[bx]              ;Segment of destination
        mov     bx,SEG_SourceIfp
        mov     ds,ds:[bx]              ;Segment of source
        mov     si,OFF_SourceIfp
        mov     di,OFF_DestIfp
        mov     ax,di                   ;Low word of return value
        mov     cx,cb
;Source in ds:si
;Destination in es:di
;Count in cx
        shr     cx,1                    ;Move by words
rep     movsw
        jnc     EvenBytes               ;Was it an even number?
        movsb                           ;Move the odd byte
EvenBytes:

;return value is destination IFP in dx:ax

cEnd
        end
```

End Listing One

Listing Two

```
/* segtable.c */

#include "windows.h"
#define GLOBAL
#include "segtable.h"

static PSEG NEAR freelist;

void FAR PASCAL SegmentInit(void)
{
        freelist = 0;
        cwSegs = MINPSEGS;
        DefineHandleTable(SegmentTable);
        segDgroup = HIWORD((void FAR *)&segDgroup);
}

PSEG FAR PASCAL SegmentAlloc(DWORD size)
{
        PSEG    pseg;
        HANDLE  handle;

        if (freelist)
        {
                if (!(handle = GlobalAlloc(GMEM_MOVEABLE, size)))
                        return((PSEG)0);

                pseg = freelist;
                freelist = (PSEG)*freelist;
        }
        else
        {
                if (cwSegs >= MAXPSEGS)
                        return((PSEG)0);

                if (!(handle = GlobalAlloc(GMEM_MOVEABLE, size)))
                        return((PSEG)0);

                pseg = &SegmentTable[cwSegs+2];
                cwSegs++;
        }
        *pseg = GlobalSegment(handle);
        return(pseg);
}

void FAR PASCAL SegmentFree(PSEG pseg)
{
#if DEBUG
        if (!VALID_VARIABLE_PSEG(pseg))
                SegmentError();
#endif
        if (*pseg)
        {
#if DEBUG
                if ((*pseg % 2) == 0)
                        SegmentError();
#endif
                GlobalFree((HANDLE)GlobalHandle(*pseg));
        }
        *(PSEG *)pseg = freelist;
        freelist = pseg;
}

void FAR PASCAL DataFree(PSEG pseg)
{
```

(continued on page 90)

Listing Two *(Listing continued, text begins on page 16.)*

```
#if DEBUG
        if (!VALIDPSEG(pseg) || (*pseg % 2) == 0)
                SegmentError();
#endif
        GlobalFree((HANDLE)GlobalHandle(*pseg));
        *pseg = 0;
}

BOOL FAR PASCAL SegmentRealloc(PSEG pseg, DWORD size)
{
        HANDLE  handle;

#if DEBUG
        if (!VALIDPSEG(pseg))
                SegmentError();
#endif
        if (*pseg)
        {
#if DEBUG
                if ((*pseg % 2) == 0)
                        SegmentError();
#endif
           return ( (BOOL) GlobalReAlloc( (HANDLE)GlobalHandle(*pseg), size,

                                GMEM_MOVEABLE));
        }
        else
        {
                if (!(handle = GlobalAlloc(GMEM_MOVEABLE, size)))
                        return(FALSE);

                *pseg = GlobalSegment(handle);
                return(TRUE);
        }
}

#if DEBUG
void FAR PASCAL SegmentError(void)
{
        FatalExit(-1);
}
#endif
```

End Listing Two

Listing Three

```
/* segtable.h */

#define MAXPSEGS        9
#define MINPSEGS        1

#define cwSegs SegmentTable[0]
#define cwClear SegmentTable[1]
#define segDgroup SegmentTable[2]

#ifndef GLOBAL
#define GLOBAL extern
#endif

typedef WORD BOOLEAN;
typedef WORD SEG;
typedef SEG NEAR *PSEG;
typedef unsigned long IFP;

#define FARPTR(off, seg) ( (void FAR *)MAKELONG(off, seg) )
#define MAKEIFP(off,pseg) ( (IFP)MAKELONG(off, pseg) )
#define IFP2SEG(ifp)            ( *(PSEG)HIWORD(ifp) )
#define IFP2PTR(ifp)    FARPTR(LOWORD(ifp), IFP2SEG(ifp))

#define GlobalSegment(handle) ((SEG) HIWORD(GlobalHandle(handle)) )

void FAR PASCAL SegmentInit(void);
PSEG FAR PASCAL SegmentAlloc(DWORD size);
void FAR PASCAL SegmentFree(PSEG pseg);
void FAR PASCAL DataFree(PSEG pseg);
BOOL FAR PASCAL SegmentRealloc(PSEG pseg, DWORD size);
void FAR PASCAL DefineHandleTable(WORD NEAR *segtable);
void FAR PASCAL SegmentError(void);

GLOBAL SEG NEAR SegmentTable[MAXPSEGS + 2];

#define VALIDPSEG(pseg) \
                (pseg > &SegmentTable[1] && \
                 pseg < &SegmentTable[MAXPSEGS + 2] && \
                 ((WORD)pseg % 2) == 0)

#define VALID_VARIABLE_PSEG(pseg) \
                (pseg > &SegmentTable[MINPSEGS + 1] && \
                 pseg < &SegmentTable[MAXPSEGS + 2] && \
                 ((WORD)pseg % 2) == 0)
```

End Listings

Listing One *(Text begins on page 28.)*

```
/* example.c - An example of three dimensional graphics using Xlib. */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <math.h>
#include <stdio.h>
#include "double_buffer.h"

typedef double Transform[4][4];
typedef double Point[3];

Display *display;           /* display connection */
Window window;              /* window identifier */
GC gc;                      /* graphics context */
XColor colors[4];           /* colors to draw with */
double_buffer_state *dbuf_state; /* state record for double buffer utilities*/
double_buffer = 1;          /* Whether to use double buffering */
unsigned int width, height; /* last known window size */
Transform model;            /* transform from world to modelling coordinates*/
Transform view;             /* transform from modelling to VDC coordinates */
Transform device;           /* transform from VDC to device coordinates */
Transform composite;        /* transform from world to device coordinates */
Transform motion;           /* transform for modelling motion */

identity(transform)
/* Set a Transform matrix to an identity matrix. */
Transform transform; /* transform to operate on */
{
        register int i, j;

        for (i = 0; i < 4; i++)
                for (j = 0; j < 4; j++)
                        transform[i][j] = (i == j);
}

rotate_X(transform, angle)
/* Set a Transform matrix to a rotation around the X axis. */
Transform transform; /* transform to operate on */
double angle;        /* angle in radians to rotate by */
{
        identity(transform);
        transform[1][1] = transform[2][2] = cos(angle);
        transform[2][1] = -(transform[1][2] = sin(angle));
}

rotate_Y(transform, angle)
/* Set a Transform matrix to a rotation around the Y axis. */
Transform transform; /* transform to operate on */
double angle;        /* angle in radians to rotate by */
{
        identity(transform);
        transform[0][0] = transform[2][2] = cos(angle);
        transform[0][2] = -(transform[2][0] = sin(angle));
}

transform_point(transform, p, tp)
/* Apply a Transform matrix to a point. */
Transform transform; /* transform to apply to the point */
Point p;             /* the point to transform */
Point tp;            /* the returned point after transformation */
{
        int i, j;
        double homogeneous[4];
        double sum;

        for (i = 0; i < 4; i++) {
                sum = 0.0;
                for (j = 0; j < 3; j++)
                        sum += p[j] * transform[j][i];
                homogeneous[i] = sum + transform[3][i];
        }

        for (i = 0; i < 3; i++)
                tp[i] = homogeneous[i] / homogeneous[3];
}

cross_product(v1, v2, c)
/* Compute the cross product of two vectors. */
Point v1, v2; /* the vectors to take the cross product of */
Point c;      /* the result */
{
        c[0] = v1[1] * v2[2] - v1[2] * v2[1];
        c[1] = v1[2] * v2[0] - v1[0] * v2[2];
        c[2] = v1[0] * v2[1] - v1[1] * v2[0];
}

backface(p1, p2, p3)
/* Determine if a polygon is a back face of an object */
Point p1, p2, p3; /* the first three vertices of a polygon */
{
        Point v1, v2, c;

        /* This relies on the first three vertices of each face being clockwise
         * around a convex angle or counter-clockwise around a concave
         * angle as viewed from the front of the face. */
        v1[0] = p2[0] - p1[0];
        v1[1] = p2[1] - p1[1];
        v1[2] = p2[2] - p1[2];
        v2[0] = p2[0] - p3[0];
        v2[1] = p2[1] - p3[1];
        v2[2] = p2[2] - p3[2];
        cross_product(v1, v2, c);
        return(c[2] > 0);
}

concatenate_transforms(transform1, transform2, result)
/* Use matrix multiplication to combine two transforms into one. */
Transform transform1, transform2; /* the transforms to combine */
Transform result;                 /* the new combined transform */
{
        register int i, j, k; /* index variables */
        Transform temporary;  /* a temporary result */
        /* Using a temporary result allows a single transform to be passed in
         * as both one of the original transforms and as the new result. */

        for (i = 0; i < 4; i++) {
                for (j = 0; j < 4; j++) {
                        temporary[i][j] = 0.0;
                        for (k = 0; k < 4; k++) {
                                temporary[i][j] += transform1[i][k]
                                                * transform2[k][j];
                        }
                }
        }

        for (i = 0; i < 4; i++)
                for (j = 0; j < 4; j++)
                        result[i][j] = temporary[i][j];
}

init_X(argc, argv)
/* Initialize the X window system */
int argc;  /* the number of program arguments */
char *argv[]; /* an array of pointers to program arguments */
{
        XEvent event;               /* holds X server events */
        static XSizeHints xsh = {           /* Size hints for window manager*/
                (PPosition | PSize | PMinSize), /* flags */
                300,                        /* height */
                300,                        /* width */
                200,                        /* minimum height */
                200,                        /* minimum width */
                5,                          /* x coordinate */
                5                           /* y coordinate */
        };
        static XWMHints xwmh = {      /* More hints for window manager */
                (InputHint | StateHint),  /* flags */
                False,                    /* input */
                NormalState,              /* initial_state */
                0,                        /* icon pixmap */
                0,                        /* icon window */
                0, 0,                     /* icon location */
                0,                        /* icon mask */
                0,                        /* Window group */
        };
        static XClassHint xch = {     /* Class hints for window manager */
                "example",                /* name */
                "EXample"                 /* class */
        };
        XGCValues gcvalues;

        if ((display = XOpenDisplay(NULL)) == NULL) {
                fprintf(stderr, "Can't open %s\n", XDisplayName(NULL));
                exit(1);
        }

        window = XCreateSimpleWindow(display,
            DefaultRootWindow(display),
            xsh.x, xsh.y,   xsh.width, xsh.height,   2,
            WhitePixel(display, DefaultScreen(display)),
            BlackPixel(display, DefaultScreen(display)));

        XSetStandardProperties(display, window, "Example", "Example",
            None, argv, argc, &xsh);
        XSetWMHints(display, window, &xwmh);
        XSetClassHint(display, window, &xch);

        XSelectInput(display, window,
            StructureNotifyMask |
            ExposureMask |
            ButtonPressMask |
            Button1MotionMask |
            PointerMotionHintMask);

        XMapWindow(display, window);
        XFlush(display);

        do {
                XNextEvent(display, &event);
        } while (event.type != MapNotify || event.xmap.window != window);

        gc = XCreateGC(display, window, 0, &gcvalues);
        XSetState(display, gc,
            WhitePixel(display, DefaultScreen(display)),
            BlackPixel(display, DefaultScreen(display)),
            GXcopy, AllPlanes);

        /* black */
        colors[0].red = 0;
        colors[0].green = 0;
        colors[0].blue = 0;

        /* white */
        colors[1].red = 65535;
        colors[1].green = 65535;
        colors[1].blue = 65535;

        /* green */
        colors[2].red = 0;
        colors[2].green = 40000;
        colors[2].blue = 0;

        /* yellow */
        colors[3].red = 65535;
        colors[3].green = 65535;
        colors[3].blue = 0;

        dbuf_state = start_double_buffer(display,
            DefaultColormap(display, DefaultScreen(display)), 2, colors);
        if (dbuf_state == NULL) {
         fprintf(stderr, "Couldn't allocate resources for double
                                                    buffering\n");
         exit(1);
```

(continued on page 94)

Listing One *(Listing continued, text begins on page 28.)*

```
        }
        XSetPlaneMask(display, gc, dbuf_state->drawing_planes);
}

init_transforms()
/* Initialize transformations for modelling, viewing, and device mapping. */
{
        Window root;
        int x, y;
        unsigned int b, d;

        identity(model);

        identity(view);
        view[2][2] = 2.0;
        view[2][3] = 1.0;
        view[3][2] = 29.0;
        view[3][3] = 15.0;

        XGetGeometry(display, window, &root, &x, &y, &width, &height, &b, &d);
        identity(device);
        device[0][0] = device[3][0] = width / 2.0;
        device[1][1] = device[3][1] = height / 2.0;

        concatenate_transforms(model, view, composite);
        concatenate_transforms(composite, device, composite);
}

#define POINTS 16     /* The total number of unique points */
#define POLYPOINTS 12 /* The maximum number of vertices in a polygon */
#define PARTIALS 6    /* The maximum number of partial polygons in a polygon*/

#define END_POLYGON -2         /* designates the end of one polygon */
#define END_POLYGONS -3        /* designates the end of all polygons */
#define END_PARTIAL_POLYGON -1 /* designates the end of a partial polygon */

redraw()
/* Draw the 3d object using the current composite transform. */
{
        static Point points[POINTS] = {
                -4.0, -4.0, -4.0,
                 4.0, -4.0, -4.0,
                -4.0,  4.0, -4.0,
                 4.0,  4.0, -4.0,
                -4.0, -4.0,  4.0,
                 4.0, -4.0,  4.0,
                -4.0,  4.0,  4.0,
                 4.0,  4.0,  4.0,
                -2.0, -4.0, -2.0,
                 2.0, -4.0, -2.0,
                -2.0,  0.0, -2.0,
                 2.0,  0.0, -2.0,
                -2.0, -4.0,  2.0,
                 2.0, -4.0,  2.0,
                -2.0,  0.0,  2.0,
                 2.0,  0.0,  2.0,
        };
        Point transformed_points[POINTS];
        static int polygons[] = {

                10, 11,  9,  8, 10, END_POLYGON,
                12, 13, 15, 14, 12, END_POLYGON,
                9, 11, 15, 13,  9, END_POLYGON,
                11, 10, 14, 15, 11, END_POLYGON,
                10,  8, 12, 14, 10, END_POLYGON,

                0,  4,  5,  1,  0, END_PARTIAL_POLYGON,
                8, 12, 13,  9,  8, END_POLYGON,

                0,  1,  3,  2,  0, END_POLYGON,
                6,  7,  5,  4,  6, END_POLYGON,
                5,  7,  3,  1,  5, END_POLYGON,
                7,  6,  2,  3,  7, END_POLYGON,
                6,  4,  0,  2,  6, END_POLYGON,

                END_POLYGONS,
        };
        XPoint buffer[POLYPOINTS]; /* a set of Xlib coordinate vertices */
        int partials[PARTIALS];    /* starting points of partial polygons */
        int num_partials;          /* number of partial polygons in a polygon*/
        int src;                   /* an index into buffer[] */
        int dest;                  /* an index into polygons[] */
        int i;                     /* an index into partials[] */
        int at_start_of_polygon;   /* flags the start of each polygon */
        int skip_polygon;          /* flags a backface polygon */

        XSetForeground(display, gc, colors[0].pixel);
        XFillRectangle(display, window, gc, 0, 0, width, height);

        for (i = POINTS - 1; i >= 0; i--)
                transform_point(composite, points[i], transformed_points[i]);

        dest = 0;
        at_start_of_polygon = True;
        partials[0] = 0;
        num_partials = 1;
        for (src = 0; polygons[src] != END_POLYGONS; src++) {
                if (at_start_of_polygon) {
                        skip_polygon = backface(
                                transformed_points[polygons[src]],
                                transformed_points[polygons[src+1]],
                                transformed_points[polygons[src+2]]);
                        at_start_of_polygon = False;
                }
                switch (polygons[src]) {
```

(continued on page 96)

3-D—USING X

Listing One *(Listing continued, text begins on page 28.)*

```
                case END_POLYGON:
                  if (!skip_polygon) {
                     XSetForeground(display, gc, colors[2].pixel);
                     XFillPolygon(display, window, gc, &buffer[0], dest,
                         Complex, CoordModeOrigin);
                     XSetForeground(display, gc, colors[1].pixel);
                     partials[num_partials] = dest;
                     for (i=0; i<num_partials; i++)
                          XDrawLines(display, window, gc, &buffer[partials[i]],
                              partials[i+1] - partials[i], CoordModeOrigin);
                  }
                  dest = 0;
                  at_start_of_polygon = True;
                  break;
             case END_PARTIAL_POLYGON:
                  partials[num_partials++] = dest;
                  break;
             default:
                  buffer[dest].x = transformed_points[polygons[src]][0];
                  buffer[dest++].y = transformed_points[polygons[src]][1];
                  break;
             }
        }
        if (double_buffer) {
             double_buffer_switch(dbuf_state);
             XSetPlaneMask(display, gc, dbuf_state->drawing_planes);
        } else {
             XSetPlaneMask(display, gc, AllPlanes);
        }
        XFlush(display);
}

main(argc, argv)
char *argv[];
{
        XEvent event;              /* holds X server events */
        int x, y;                  /* the last X pointer position */
        int new_x, new_y;          /* a new X pointer position */
        unsigned int mask;         /* mask of button and modifier key state */
        unsigned int dummy;        /* placeholder for unwanted return values */

        init_X(argc, argv);
        init_transforms();

        printf("Drag button 1 to rotate the object.\n");
        printf("Press button 2 to toggle double buffering on and off.\n");
        printf("Press button 3 to stop the program.\n");

        for (; ; ) {
                XNextEvent(display, &event);
                switch (event.type) {
                case DestroyNotify:
                        XCloseDisplay(display);
                        exit(0);
                        break;
                case Expose:
                        if (event.xexpose.count == 0) {
                                redraw();
                        }
                        break;
                case ConfigureNotify:
                   if ((event.xconfigure.width != width) ||
                        (event.xconfigure.height != height)) {
                        width = event.xconfigure.width;
                        height = event.xconfigure.height;

                        device[0][0] = device[3][0] = width / 2.0;
                        device[1][1] = device[3][1] = height / 2.0;

                        concatenate_transforms(model, view, composite);
                        concatenate_transforms(composite, device, composite);
                        redraw();
                   }
                   break;
                case MotionNotify:
                        XQueryPointer(display, window,
                            &dummy, &dummy,
                            &dummy, &dummy,
                            &new_x, &new_y,
                            &mask);
                        if (!(mask & Button1Mask))
                                break;
                        rotate_X(motion, M_PI / 360.0 * (new_y - y));
                        concatenate_transforms(model, motion, model);
                        rotate_Y(motion, -M_PI / 360.0 * (new_x - x));
                        concatenate_transforms(model, motion, model);
                        x = new_x;
                        y = new_y;
                        concatenate_transforms(model, view, composite);
                        concatenate_transforms(composite, device, composite);
                        redraw();
                        break;
                case ButtonPress:
                        x = event.xbutton.x;
                        y = event.xbutton.y;
                        if (event.xbutton.button == 2)
                                double_buffer ^= 1;
                        if (event.xbutton.button == 3)
                                exit(0);
                        break;
                default:
                        break;
                }
        }
}
```

End Listing One

(Listings continued on page 98)

Listing Two *(Listing continued, text begins on page 28.)*

```
/* double_buffer.h - declarations for an Xlib double buffering utility. */

/* double buffering state record */
typedef struct {
        Display *display;
        Colormap cmap;
        long drawing_planes;     /* planes currently drawn to */
        int buffer;              /* which buffer to show, even or odd */
        XColor *colormaps[2];    /* color maps for even and odd buffers */
        int map_size;            /* number of entries in color maps */
        long masks[2];           /* write_enable masks for odd and even */
        long *planes;            /* individual planes */
        long pixel;              /* pixel base value of double buffering */
} double_buffer_state;

/* double buffering procedures */
extern double_buffer_state *start_double_buffer();
extern void double_buffer_switch();
extern void end_double_buffer();
```

End Listing Two

Listing Three

```
/* double_buffer.c - an Xlib double buffering utility. */

#include <X11/Xlib.h>
#include <malloc.h>
#include <stdio.h>
#include "double_buffer.h"

static void release(state)
register double_buffer_state *state;
/* Release a possibly partially allocated double buffer state record. */
{
        if (state != NULL) {
                if (state->colormaps[0] != NULL) free(state->colormaps[0]);
                if (state->colormaps[1] != NULL) free(state->colormaps[1]);
                if (state->planes != NULL) free(state->planes);
                free(state);
        }
}

static long color(state, simple_color)
register double_buffer_state *state;
register long simple_color;
/* Map the supplied color into the equivalent color
 * using the double buffered planes. */
{
        register long i, plane, computed_color;

        computed_color = state->pixel;
        for (plane = 1, i = 0; simple_color != 0; plane <<= 1, i++) {
                if (plane & simple_color) {
                        computed_color |= state->planes[i];
                        simple_color &= ~plane;
                }
        }
        return(computed_color);
}

double_buffer_state *start_double_buffer(display, cmap, planes, colors)
Display *display;
Colormap cmap;
long planes;    /* how many planes for each buffer */
XColor *colors; /* color settings for buffers */
/* Start double buffering in given number of planes per buffer.
 * If resources can be allocated, then set color pixels in colors parameter
 * and return the address of a double_buffer_state record.
 * Otherwise, return NULL.  */
{
        register double_buffer_state *state;
        register long i, high_mask, low_mask;

        /* Allocate memory. */
        state = (double_buffer_state *) malloc(sizeof(double_buffer_state));
        if (state == NULL)
                return (NULL);

    state->map_size = 1 << (2 * planes);
    state->colormaps[0] = (XColor *) malloc(state->map_size *sizeof(XColor));
    state->colormaps[1] = (XColor *) malloc(state->map_size *sizeof(XColor));
    state->planes = (long *) malloc((2 * planes) * sizeof(long));
    if (state->colormaps[1] == NULL || state->colormaps[0] == NULL
                || state->planes == NULL) {
                release(state);
                return(NULL);
    }
    state->display = display;
    state->cmap = cmap;

        /* Get colors to double buffer with. */
        if (XAllocColorCells(state->display, state->cmap, False,
                state->planes, 2*planes, &state->pixel, 1) == 0) {
                release(state);
                return(NULL);
        }

        /* Prepare the write enable masks. */
        state->masks[0] = AllPlanes;
        state->masks[1] = AllPlanes;
        /* Mask 0 won't write in the "low" planes. */
        /* Mask 1 won't write in the "high" planes. */
        for (i = 0; i < planes; i++) {
                state->masks[0] &= ~state->planes[i];
                state->masks[1] &= ~state->planes[planes + i];
        }

        /* Prepare the flags and pixel values for each color. */
        for (i = 0; i < (1 << planes); i++) {
                colors[i].pixel = color(state, i | (i << planes));
                colors[i].flags = DoRed | DoGreen | DoBlue;
        }

        /* Prepare the two color map settings. */
        /* Colormap 0 displays the "low" planes. */
        /* Colormap 1 displays the "high" planes. */
        low_mask = (1 << planes) - 1;
        high_mask = low_mask << planes;
        for (i = state->map_size - 1; i >= 0; i--) {
                state->colormaps[0][i] = colors[i & low_mask];
                state->colormaps[0][i].pixel = color(state, i);

                state->colormaps[1][i] = colors[(i & high_mask) >> planes];
                state->colormaps[1][i].pixel = color(state, i);
        }

        /* Set up initial color map and write_enable. */
        state->buffer = 0;
        state->drawing_planes = state->masks[state->buffer];
        XStoreColors(state->display, state->cmap,
                state->colormaps[state->buffer], state->map_size);

        return(state);
}

void double_buffer_switch(state)
register double_buffer_state *state;
/* Change double buffering buffer.
 * Return the new planes mask for double buffering. */
{
        /* Toggle the buffers. */
        state->buffer ^= 1;

        /* Adjust the color map and write enable mask. */
        XStoreColors(state->display, state->cmap,
                state->colormaps[state->buffer], state->map_size);

        state->drawing_planes = state->masks[state->buffer];
}

void end_double_buffer(state)
register double_buffer_state *state;
{
        XFreeColors(state->display, state->cmap,
                &state->pixel, 1, ~(state->masks[0] & state->masks[1]));
        release(state);
}
```

End Listings

Listing One *(Text begins on page 38.)*

```
MODULE Mtest ;
FROM MENU IMPORT MenuType,Menu ;
FROM InOut IMPORT WriteString, WriteLn ;
PROCEDURE CLS ;
BEGIN
        WriteString(CHR(12)) ;
END CLS ;

PROCEDURE Header ;
BEGIN
        WriteLn ;
        WriteLn ;
        WriteLn ;
        WriteLn ;
        WriteString('          M A I L I N G   L I S T   M E N U');
        WriteLn ;
        WriteString('                        (Press Esc key to leave progam)') ;
END Header ;

PROCEDURE EnterData ;
END EnterData ;
PROCEDURE PrintZip ;
END PrintZip ;
PROCEDURE Modify ;
END Modify ;
PROCEDURE DelData ;
END DelData ;
PROCEDURE Browse ;
END Browse ;
PROCEDURE Backup ;
END Backup ;
PROCEDURE PrintNZip ;
END PrintNZip ;
PROCEDURE Setup ;
END Setup ;
PROCEDURE test ;
VAR
        test    : MenuType ;
        i       : CARDINAL ;
BEGIN
        LOOP
                CLS ;
                Header ;
                test[0] := '1. Enter new mail list data.' ;
                test[1] := '2. Print a Zip code sorted mail list.' ;
                test[2] := '3. Change existing data.' ;
                test[3] := '4. Delete a single address from the list.' ;
                test[4] := '5. Browse through the existing data.' ;
                test[5] := '6. Backup data.' ;
                test[6] := '7. Print a mail list not sorted by ZIP.' ;
                test[7] := '8. Perform other functions.' ;
                i := Menu(test,8) ;
                CASE i OF
                        0 : EXIT      |
                        1 : EnterData |
                        2 : PrintZip  |
                        3 : Modify    |
                        4 : DelData   |
                        5 : Browse    |
                        6 : Backup    |
                        7 : PrintNZip |
                        8 : Setup
                END ; (* CASE *)
        END ; (* LOOP *)
END test ;
BEGIN
        test ;
END Mtest.
```

End Listing One

Listing Two

```
DEFINITION MODULE READA;
(* EscType determines whether an esc will exit a field. The values are:
        Esc    which allows an escape to exit a field.
        NoEsc which prevents exit from a field on an escape char.*)
FROM SYSTEM IMPORT AX,BX,CX,DX,BP,CODE,SETREG ;
FROM Terminal IMPORT Write ;
FROM InOut IMPORT EOL ;
        EXPORT QUALIFIED Grab, ClearField, gotoxy, EscType ;
TYPE
     EscType = (Esc,NoEsc) ;
     PROCEDURE Grab(VAR String : ARRAY OF CHAR ; EscFlag :EscType) ;
     PROCEDURE ClearField(VAR String : ARRAY OF CHAR ; Column,Row : CARDINAL);
     PROCEDURE gotoxy(x,y : CARDINAL ) ;
END READA.
```

End Listing Two

Listing Three

```
(*****************************************************************************
        Name: READA
        Purpose: Usefull String routines
                ClearField wipes out a data entry field on screen
                Grab accepts characters up to length of length of string array
                then refuses to accept any more chars until Enter is pressed.
                gotoxy positions the cursor.
        Entry: ClearField(VAR String: ARRAY OF CHAR ; Column,Row : CARDINAL)
               Grab(VAR String:ARRAY OF CHAR ; EscFlag : EscType)
               gotoxy(x,y : CARDINAL) x = column y = row.
        Exit: ClearField - String is zeroed, cursor left at position
```

```
                                                        Column,Row
            Grab - String is filled in with user entered characters.
        Global Variables used: Passed String array.
        Revision number:
        1.2   10/3/88   Escape type added to Grab.
        1.1   11/30/87  Escape key exit for String 0.
        1.0   11/8/87
*****************************************************************************)

IMPLEMENTATION MODULE READA ;
FROM SYSTEM IMPORT AX,BX,CX,DX,BP,CODE,SETREG ;
FROM Terminal IMPORT Write, Read ;
FROM InOut IMPORT EOL ;
VAR
        Index   : CARDINAL ;
        Ch      : CHAR ;
PROCEDURE gotoxy(x,y : CARDINAL ) ;
VAR
        a : CARDINAL ;
BEGIN
        IF ( x >= 0) AND ( x <= 79) AND ( y >=0) AND ( y <=24) THEN
                IF ( x # 79) OR (y # 24 )
                THEN
                CODE( 55H) ;            (* PUSH BP *)
                a := 200H ;
                SETREG ( AX ,a ) ;
                CODE( 50H ) ;           (* PUSH AX *)
                a := 0H ;
                SETREG( BX , a) ;
                CODE( 53H) ;            (* PUSH BX *)
                SETREG( DX,x + 256 * y) ;
                CODE( 5BH ) ;           (* POP BX *)
                CODE( 58H ) ;           (* POP AX *)
                CODE( 0CDH,10H) ;       (* INT 10H *)
                CODE( 5DH ) ;           (* POP BP *)
                END ;
        END ;
END gotoxy ;

PROCEDURE ClearField(VAR String : ARRAY OF CHAR ; Column,Row : CARDINAL) ;
(* This procedure wipes the appropriate field on the screen out *)
BEGIN
        gotoxy(Column,Row) ;    (* Position Cursor *)
        FOR Index := 0 TO HIGH(String) DO
                Write(' ') ;
        END ;   (* FOR *)
        gotoxy(Column,Row) ;    (* Reposition Cursor *)
END ClearField ;

PROCEDURE Grab(VAR String : ARRAY OF CHAR ; EscFlag : EscType ) ;

(* This procedure assumes that the cursor has already been moved to a position
either by a direct gotoxy call or by a call to ClearField *)

BEGIN
        FOR Index := 0 TO HIGH(String) DO
                String[Index] := CHR(0) ;
        END ; (* FOR *)
        Index := 0 ;
        LOOP
                Read(Ch) ;
                IF Ch = EOL THEN EXIT END ;
                IF EscFlag = Esc THEN
                        IF Ch = CHR(27) THEN
                                String[0] := Ch ;
                                EXIT ;
                        END ;
                END ;
                IF Ch = CHR(8) THEN
                        IF Index = 0 THEN
                                Write(CHR(7)) ; (* Honk at Barney *)
                        ELSE
                                Write(CHR(8)) ; (* BackSpace *)
                                Write(CHR(32)) ;        (* Space *)
                                Write(CHR(8)) ; (* BackSpace *)
                                Index := Index - 1 ;
                                String[Index] := CHR(0) ;
                        END ; (* IF *)
                ELSIF Ch < CHR(32) THEN
                        Write(CHR(7)) ; (* Honk at Barney *)
                ELSE
                        IF Index = (HIGH(String) +1) THEN
                                Write(CHR(7)) ;
                        ELSE
                                String[Index] := Ch ;
                                Write(Ch) ;
                                Index := Index + 1 ;
                        END ; (* IF *)
                END ; (* IF *)
        END ; (* LOOP *)
END Grab ;
END READA .
```

End Listing Three

Listing Four

```
DEFINITION MODULE MENU ;
  EXPORT QUALIFIED MenuType,Menu ;
  TYPE MenuType = ARRAY[0..59],[0..79] OF CHAR ;
  PROCEDURE Menu(VAR A : MenuType ; NumberOfMenuEntries : CARDINAL) :CARDINAL
;
END MENU.
```

End Listing Four

(Listings continued on page 102)

Listing Five *(Listing continued, text begins on page 38.)*

```
(*****************************************************************************
Name: MENU
Purpose: Automatic screen layout, and response error checking for
   Pick-a-number menus.
Entry: Menu(VAR A : MenuType ; NumberOfMenuEntries : CARDINAL): CARDINAL ;
Exit: Qualified acceptance of menu item or escape key.
Revision Number:
1.1   10/3/88 Escape key output changed to = 0
1.0   9/26/88
*****************************************************************************)

IMPLEMENTATION MODULE MENU ;
FROM READA IMPORT gotoxy, Grab,EscType ;
FROM Strings IMPORT Length ;
FROM NumberConversion IMPORT StringToCard ;
FROM InOut IMPORT WriteString ;

PROCEDURE OneColumn(VAR A : MenuType ; i : CARDINAL) ;
VAR
        j,k,l,m : CARDINAL ;
BEGIN
        i := i - 1 ; (* Convert from one base to zero based *)
        (* First we center the strings to be displayed vertically *)
        j := (5 + ((15 - i) DIV 2)) ;
        (* Now we center the strings horizontally *)
        l := 0 ;
        FOR m := 0  TO i DO
                k := Length(A[m]) ;
                IF (k > l) THEN l := k END ; (* get longest string length *)
        END ; (* FOR *)
        k := (40 -(l DIV 2)) ;

        (* Now print the menu *)
        FOR m := 0 TO i DO
                gotoxy(k,(j+m)) ; (* Position cursor to string position *)
                WriteString(A[m]) ;
        END ; (* FOR *)
END OneColumn ;

PROCEDURE TwoColumns(VAR A : MenuType ; i : CARDINAL) ;
VAR
        j,k,l,m,n,o,p : CARDINAL ;
BEGIN
        (* First we center the strings to be displayed vertically *)
        i := i - 1 ;    (* Convert from one base to zero based *)
        n := i DIV 2 ;
        j := (5 + ((15 - n) DIV 2)) ;
        (* Now we center the strings horizontally *)
        l := 0 ;
        FOR m := 0  TO n-1 DO
                k := Length(A[m]) ;
                IF (k > l) THEN l := k END ; (* get longest string length *)
        END ; (* FOR *)
        k := (20 -(l DIV 2)) ;
(* Now set up the second column centered on position 60 *)
        o := 0 ;
        FOR m := n   TO i DO
                p := Length(A[m]) ;
                IF (p > o) THEN o := p END ; (* get longest string length *)
        END ; (* FOR *)
        p := (60 -(o DIV 2)) ;

        (* Now print the menu *)
        FOR m := 0 TO n-1 DO
                gotoxy(k,(j+m)) ; (* Position cursor to string position *)
                WriteString(A[m]) ;
        END ; (* FOR *)
        FOR m := n TO i DO
                gotoxy(p,(j+m-(n))) ; (* Position cursor to string position *)
                WriteString(A[m]) ;
        END ; (* FOR *)

END TwoColumns ;

PROCEDURE ThreeColumns(VAR A : MenuType ; i : CARDINAL) ;
VAR
        j,k,l,m,n,o,p,q,r : CARDINAL ;
BEGIN
        (* First we center the strings to be displayed vertically *)
        i := i - 1 ;    (* Convert from one base to zero based *)
        n := i DIV 3 ;
        j := i MOD 3 ;
        IF j = 2 THEN INC(n) END ;
        j := (5 + ((15 - n) DIV 2)) ;
        (* Now we center the strings horizontally *)
        l := 0 ;
        FOR m := 0  TO n-1 DO
                k := Length(A[m]) ;
                IF (k > l) THEN l := k END ; (* get longest string length *)
        END ; (* FOR *)
        k := (20 -(l DIV 2)) ;
(* Now set up the second column centered on position 40 *)
        o := 0 ;
        FOR m := n  TO (2*n)-1 DO
                p := Length(A[m]) ;
                IF (p > o) THEN o := p END ; (* get longest string length *)
        END ; (* FOR *)
        p := (40 -(o DIV 2)) ;
(* Now set up the third column centered on position 60 *)
        q := 0 ;
        FOR m := 2*n   TO i DO
                r := Length(A[m]) ;
                IF (r > q) THEN q := r END ; (* get longest string length *)
        END ; (* FOR *)
        r := (60 -(q DIV 2)) ;

        (* Now print the menu *)
        FOR m := 0 TO n-1 DO
                gotoxy(k,(j+m)) ; (* Position cursor to string position *)
                WriteString(A[m]) ;
        END ; (* FOR *)
        FOR m := n TO 2*n-1 DO
                gotoxy(p,(j+m-n)) ; (* Position cursor to string position *)
                WriteString(A[m]) ;
        END ; (* FOR *)
        FOR m := 2*n TO i DO
                gotoxy(r,(j+m-2*n)) ; (* Position cursor to string position *)
                WriteString(A[m]) ;
        END ; (* FOR *)
END ThreeColumns ;

PROCEDURE FourColumns(VAR A : MenuType ; i : CARDINAL) ;
VAR
        j,k,l,m,n,o,p,q,r,s,t : CARDINAL ;
BEGIN
        (* First we center the strings to be displayed vertically *)
        i := i - 1 ;    (* Convert from one base to zero based *)
        n := i DIV 4 ;
        j := i MOD 4 ;
        IF j = 3 THEN INC(n) END ;
        j := (5 + ((15 - n) DIV 2)) ;
        (* Now we center the strings horizontally *)
        l := 0 ;
        FOR m := 0  TO n-1 DO
                k := Length(A[m]) ;
                IF (k > l) THEN l := k END ; (* get longest string length *)
        END ; (* FOR *)
        k := (16 -(l DIV 2)) ;
(* Now set up the second column centered on position 40 *)
        o := 0 ;
        FOR m := n   TO 2*n-1 DO
                p := Length(A[m]) ;
                IF (p > o) THEN o := p END ; (* get longest string length *)
        END ; (* FOR *)
        p := (32 -(o DIV 2)) ;
(* Now set up the third column centered on position 60 *)
        q := 0 ;
        FOR m := 2*n   TO 3*n-1 DO
                r := Length(A[m]) ;
                IF (r > q) THEN q := r END ; (* get longest string length *)
        END ; (* FOR *)
        r := (48 -(q DIV 2)) ;
        s := 0 ;
        FOR m := 3*n   TO i DO
                t := Length(A[m]) ;
                IF (t > s) THEN s := t END ; (* get longest string length *)
        END ; (* FOR *)
        t := (64 -(s DIV 2)) ;

        (* Now print the menu *)
        FOR m := 0 TO n-1 DO
                gotoxy(k,(j+m)) ; (* Position cursor to string position *)
                WriteString(A[m]) ;
        END ; (* FOR *)
        FOR m := n TO 2*n-1 DO
                gotoxy(p,(j+m-(n))) ; (* Position cursor to string position *)
                WriteString(A[m]) ;
        END ; (* FOR *)
        FOR m := 2*n TO 3*n-1 DO
                gotoxy(r,(j+m-(2*n))) ; (* Position cursor to string position *)
                WriteString(A[m]) ;
        END ; (* FOR *)
        FOR m := 3*n TO i DO
                gotoxy(t,(j+m-(3*n))) ; (* Position cursor to string position *)
                WriteString(A[m]) ;
        END ; (* FOR *)
END FourColumns ;

PROCEDURE Menu(VAR A : MenuType ; NumberOfMenuEntries : CARDINAL): CARDINAL ;
VAR
        i,j,k,l : CARDINAL ;
        input   : ARRAY[0..1] OF CHAR ;
        done    : BOOLEAN ;
BEGIN
(* 'A' is actually an array of character strings ( an array of array of char)
Menu displays 'A' and waits for up to a two character response with a trailing
carriage return. Menu returns 100 if escape is pressed, otherwise returns
number entered by user as menu response.(0..60).*)

        i := NumberOfMenuEntries ;
        IF (i <= 15 ) THEN OneColumn(A,i) END ;
        IF ((i > 15) AND (i <= 30)) THEN TwoColumns(A,i) END ;
        IF ((i > 30) AND (i <= 45 )) THEN ThreeColumns(A,i) END ;
        IF (i > 45) THEN FourColumns(A,i) END ;
        (* Allow a maximum of 15 items per column on displayed menu.*)
  LOOP
    gotoxy(5,24) ;
    WriteString('Enter the number of your selection and press Enter key:   ') ;
    WriteString(CHR(08)) ;
    WriteString(CHR(08)) ;
    Grab(input,Esc) ;
(* If Esc is pressed instead of a number exit with an impossible  value *)
                IF (input[0] = CHR(27)) THEN RETURN 0 END ;
                StringToCard(input,j,done) ;
(* Return only legal values of input *)
                IF done THEN
                        IF (j > 0) AND ( j <=i) THEN RETURN j END ;
                END ; (* IF *)
        END ; (* LOOP *)
END Menu ;
END MENU .
```

End Listings

Listing One *(Text begins on page 44.)*

```
{*** Singly linked move-to-the front list ***}
{*** Contents: "LInsert", "Mtffind" ***}

{ Data Structure:
  ptr=^node;
  node=RECORD rec:item; next:ptr; END;  }

PROCEDURE LInsert(arg:item; VAR root:ptr);
    VAR p:ptr;                          { To generate storage }
    BEGIN
      NEW(p);                           { Allocate }
      p^.rec:=arg;                      { Add data }
      p^.next:=root;                    { Place at front of list }
      root:=p;                          { Point to new front of list }
    END;

FUNCTION Mtffind(arg:item; VAR root:ptr;):boolean;
    VAR temp1,temp2:ptr;                { Search pointers }
        found:boolean;                  { TRUE if  found }
    BEGIN
      temp1:=root;                      { Get a copy of starting location }
      temp2:=root;                      { Secondary copy }
      found:=false;                     { Nothing found yet }

      WHILE (temp1<>NIL) AND (NOT found) DO
          BEGIN
            IF temp1^.rec<>arg THEN     { Found it? }
                BEGIN                   { Nope... }
                  temp2:=temp1;         { Move trailing pointer }
                  temp1:=temp1^.next;   { Move search pointer }
                END
            ELSE found:=true;           { Yup... }
          END;

      IF found THEN                     { Move item to front of list }
          BEGIN
            temp2^.next:=temp1^.next;
            IF temp1<>root THEN temp1^.next:=root;
            root:=temp1;
          END;

      Mtffind:=found;
    END;
```

End Listing One

Listing Two

```
{*** Move To The Front Splay Tree ***}
{*** Contents: SplaySearch, BSInsert, BSDelete ***}

{ Data Structure:
  ptr=^node;
  node=RECORD data:key; left,right:ptr; END;    }

FUNCTION SplaySearch(x:key; VAR p:ptr):boolean;
TYPE noderec=RECORD                     { Temporary Tree Pointer Def. }
               left,right:ptr;
             END;
VAR l,r:noderec;                        { Temporary Trees }
    done:boolean;                       { TRUE if NIL encountered in search }

  PROCEDURE RRot(VAR p:ptr);
    VAR temp,temp1:ptr;                 { Temporary pointers }
    BEGIN
      IF p<>NIL THEN                    { Don't rotate if nothing's there }
        IF p^.left<>NIL THEN            { No left edge - don't rotate }
          BEGIN
            temp:=p; temp1:=p^.left^.right;  { Copy root & 2ndary child }
            p:=temp^.left; p^.right:=temp;   { Rotate root }
            temp^.left:=temp1;          { Reattach 2ndary child }
          END;
    END;

  PROCEDURE LRot(VAR p:ptr);
    VAR temp,temp1:ptr;                 { Temporary pointers }
    BEGIN
      IF p<>NIL THEN                    { Don't rotate if nothing's there }
        IF p^.right<>NIL THEN           { No right edge - don't rotate }
          BEGIN
            temp:=p; temp1:=p^.right^.left;  { Copy root & 2ndary child }
            p:=temp^.right; p^.left:=temp;   { Rotate root }
            temp^.right:=temp1;         { Reattach 2ndary child }
          END;
    END;

  PROCEDURE LnkRight(VAR p:ptr; VAR r:noderec);
    VAR temp:ptr;                       { Temporary pointer }
    BEGIN
      IF p^.left<>NIL THEN              { No left child - don't cut & link }
        BEGIN
          temp:=p^.left; p^.left:=NIL;  { Remember left child & break link }
          IF r.left=NIL THEN            { Attach to temporary tree }
            BEGIN r.left:=p; r.right:=p;END { Empty tree? }
          ELSE                          { Just add to bottom leftmost }
            BEGIN r.right^.left:=p; r.right:=r.right^.left; END;
          p:=temp;                      { New root is left child }
        END;
    END;
```

(Listing continued on next page)

Listing Two *(Listing continued, text begins on page 44.)*

```
PROCEDURE LnkLeft(VAR p:ptr; VAR l:noderec);
  VAR temp:ptr;                          { Temporary pointer }
  BEGIN
    IF p^.right<>NIL THEN                { No right child - don't cut & link }
      BEGIN
        temp:=p^.right; p^.right:=NIL;{ Remember right child & break link }
        IF l.left=NIL THEN               { Attach to temporary tree }
          BEGIN l.left:=p; l.right:=p;END { Empty tree? }
        ELSE                             { Just add to bottom rightmost }
          BEGIN l.right^.right:=p; l.right:=l.right^.right; END;
        p:=temp;                         { New root is right child }
      END;
  END;

PROCEDURE Assemble(VAR p:ptr; VAR l,r:noderec);
  VAR temp,temp1:ptr;
  BEGIN
    temp:=p^.left; temp1:=p^.right;   { Hold onto subtrees }
    IF l.left<>NIL THEN
      BEGIN
        p^.left:=l.left;              { Attach temporary left subtree }
        l.right^.right:=temp;         { Reattach orginal left subtree }
      END;
    IF r.left<>NIL THEN
      BEGIN
        p^.right:=r.left;             { Attach temporary right subtree }
        r.right^.left:=temp1;         { Reattach original right subtree }
      END;
  END;

BEGIN
  l.left:=NIL; l.right:=NIL;          { Initialize temp trees }
  r.left:=NIL; r.right:=NIL;
  done:=false;                        { Init to "item maybe there" }
  IF p<>NIL THEN                      { No search if tree's empty }
    BEGIN
    REPEAT
      IF (x<p^.data) THEN             { Item on left subtree? }
        IF (p^.left<>NIL) THEN
          BEGIN
            IF x=p^.left^.data THEN LNKRIGHT(p,r)
            ELSE
            IF x<p^.left^.data THEN BEGIN RRot(p); LNKRIGHT(p,r); END
            ELSE
            IF x>p^.left^.data THEN BEGIN LNKRIGHT(p,r);LNKLEFT(p,l);END;
          END ELSE done:=TRUE
      ELSE
      IF (x>p^.data) THEN             { Item on right subtree? }
        IF (p^.right<>NIL) THEN
          BEGIN
            IF x=p^.right^.data THEN LNKLEFT(p,l)
            ELSE
            IF x>p^.right^.data THEN BEGIN LRot(p); LNKLEFT(p,l); END
            ELSE
            IF x<p^.right^.data THEN BEGIN LNKLEFT(p,l);LNKRIGHT(p,r);END;
          END ELSE done:=TRUE;
    UNTIL (x=p^.data) OR DONE;
    ASSEMBLE(p,l,r); SplaySearch:=(x=p^.data);
  END ELSE SplaySearch:=FALSE;
END;

PROCEDURE BSInsert(x:key; VAR root:ptr);
VAR p:ptr;
BEGIN
  NEW(p);
  p^.data:=x;
  p^.left:=NIL; p^.right:=NIL;
  IF root=NIL THEN root:=p                    { No tree, just insert }
  ELSE
  BEGIN
    IF NOT SplaySearch(x,root) THEN           { Is it already there? }
      IF x<root^.data THEN                    { Less than? }
        BEGIN
          p^.right:=root;                     { Root item greater than }
          p^.left:=root^.left;                { Link up left child }
          root^.left:=NIL; root:=p;           { Break link; root=new item }
        END
      ELSE
      IF x>root^.data THEN                    { Greater than? }
        BEGIN
          p^.left:=root;                      { Root item less than }
          p^.right:=root^.right;              { Link up right child }
          root^.right:=NIL; root:=p;          { Break link; root=new item }
        END;
  END;
END;

{*** Move To The Fron Splay Tree ***}
{*** Contents:SplaySearch, BSInsert, BSDelete ***}

PROCEDURE BSDelete(x:key; VAR root:ptr);
CONST maxkey=MAXINT;                          { maxkey user example
VAR temp1,temp2,temp4:ptr;
    temp3:key;
    flg:boolean;
BEGIN
  IF SplaySearch(x,root) THEN
    BEGIN
      temp1:=root^.left; temp2:=root^.right;  { Save subtrees }
      IF temp1<>NIL THEN                      { Is there a left subtree? }
        BEGIN
          flg:=SplaySearch(temp3,temp1);

          temp1^.right:=temp2;                { Attach right subtree }
        END ELSE temp1:=temp2;                { Just attach right tree }
       dispose(root);
       root:=temp1;                           { Return new tree }
    END;
END;
```

End Listing Two

Listing Three

```
{*** Self-adjusting heap ***}
{*** Contents: Merge, Min, Insert, DeleteMin routines ***}

{ Data Structure:
  ptr=^node;
  node=RECORD data:item; left,right:ptr; END;   }

FUNCTION Merge(q1,q2:ptr):ptr;
  TYPE Qrec=RECORD
              front,rear:ptr;
            END;
  VAR Q:Qrec;
  PROCEDURE Enqueue(VAR q1:ptr; VAR Q:Qrec);
    VAR temp:ptr;
    BEGIN
      temp:=q1;                           { Save top of heap }
      q1:=q1^.right;                      { Point to next top of heap }
      temp^.right:=temp^.left;            { Swap right child to left }
      temp^.left:=NIL;                    { Make sure left link's broken }
      IF q.front=NIL THEN                 { Empty merge queue }
        BEGIN
          q.front:=temp; q.rear:=temp;
        END
      ELSE                                { Oops, just add to last leftchild }
        BEGIN
          q.rear^.left:=temp; q.rear:=temp;
        END;
    END;
  BEGIN
    q.front:=NIL; q.rear:=NIL;            { Init merge queue }
    WHILE (q1<>NIL) AND (q2<>NIL) DO      { Pairwise compare and merge }
        IF q1^.data<=q2^.data THEN Enqueue(q1,q)
        ELSE Enqueue(q2,q);

    IF (q1<>NIL) AND (q2=NIL) THEN
      BEGIN
        IF q.rear<>NIL THEN q.rear^.left:=q1
        ELSE q.front:=q1;
      END
    IF (q1=NIL) AND (q2<>NIL) THEN
      BEGIN
        IF q.rear<>NIL THEN q.rear^.left:=q2
        ELSE q.front:=q2;
      END;
    Merge:=q.front;
  END;

FUNCTION Min(q1:ptr; VAR x:ptr):boolean;
  BEGIN
    x:=q1;
    Min:=(q1<>NIL);
  END;

PROCEDURE Insert(x:item; VAR q:ptr);
  VAR p:ptr;
  BEGIN
    NEW(p);                               { Allocate }
    p^.data:=x;                           { Fill it! }
    p^.left:=NIL; p^.right:=NIL;          { No children }
    q:=Merge(q,p);                        { Add it to heap }
  END;

FUNCTION DeleteMin(q:ptr; VAR x:ptr):ptr;
  BEGIN
    IF Min(q,x) THEN                      { Is there a min to delete? }
      DeleteMin:=Merge(q^.left,q^.right)
    ELSE DeleteMin:=NIL;                  { Nothing at all }
  END;

{ Pairing Heaps as described by Tarjan, et al from Algorithmica:
  Data Structure:
  TYPE hptr=^node;
       node=RECORD
              wt:integer;
              parent,left,right:hptr;
            END;                          }

FUNCTION Merge(arg1,arg2:hptr):hptr;
BEGIN
  IF (arg1<>NIL) AND (arg2<>NIL) THEN           { 2 Queues to merge? }
    BEGIN
      IF arg1^.wt<arg2^.wt THEN                 { Which is minimal? }
        BEGIN
          arg2^.parent:=arg1;                   { Who's the parent? }
          arg2^.right:=arg1^.left;              { Point to arg1's child }
          arg1^.left:=arg2;                     { It's officially a child }
          Merge:=arg1;
        END
      ELSE
        BEGIN
          arg1^.parent:=arg2;                   { Who's the parent? }
          arg1^.right:=arg2^.left;              { Point to arg2's child }
          arg2^.left:=arg1;                     { It's officially a child }
          Merge:=arg2;
        END;
    END
  ELSE
  IF (arg1<>NIL) THEN Merge:=arg1               { Just arg1's queue }
  ELSE Merge:=arg2                              { Anything else }
END;

PROCEDURE Insert(a1,a2,x:integer; VAR root:hptr);
VAR p:hptr;
BEGIN
  New(p);                                       { Allocate }
  p^.v1:=a1; p^.v2:=a2;
  p^.wt:=x;  p^.parent:=NIL;                    { Set key }
  p^.left:=NIL; p^.right:=NIL;                  { Set pointers }
```

```
  root:=Merge(p,root);                          { Add it... }
END;

FUNCTION Min(root:hptr; VAR minitem:hptr):boolean;
BEGIN
  minitem:=root;                                { What's at the root? }
  Min:=(minitem<>NIL);                          { Anything there? }
END;

FUNCTION DeleteMin(root:hptr; VAR minitem:hptr):hptr;
VAR arg1,arg2,p1:hptr;
BEGIN
  IF Min(root,minitem) THEN
    BEGIN
      root:=NIL;                                { ReInit root }
      p1:=minitem^.left;                        { Save kids }
      WHILE p1<>NIL DO                          { For all subtrees }
        BEGIN
          arg1:=p1;                             { First Subtree }
          p1:=p1^.right;                        { Move along }
          arg2:=p1;                             { Next potential subtree }
          IF p1<>NIL THEN p1:=p1^.right;        { If not NIL, move on }
          root:=Merge(Merge(arg1,arg2),root);   { Merge result with current }
        END;
      IF root<>NIL THEN root^.right:=NIL;
      DeleteMin:=root;
    END ELSE DeleteMin:=NIL;
END;

FUNCTION LinkSearch(p:hptr):hptr;
VAR temp:hptr;
BEGIN
  temp:=p^.parent^.left;
  WHILE (temp<>p) AND (temp^.right<>p) AND (temp^.right<>NIL) DO
    temp:=temp^.right;
  LinkSearch:=temp;
END;

FUNCTION DecreaseKey(change:integer; p,root:hptr):hptr;
VAR temp:hptr;
BEGIN
  IF (p<>NIL) AND (root<>NIL) THEN
    BEGIN
      p^.wt:=p^.wt-ABS(change);
      IF p=root THEN DecreaseKey:=root
      ELSE
      BEGIN
        temp:=LinkSearch(p);
        IF temp=p THEN p^.parent^.left:=p^.parent^.left^.right
        ELSE temp^.right:=p^.right;
        DecreaseKey:=Merge(p,root);
      END;
    END;
END;

FUNCTION Delete(p,root:hptr):hptr;
VAR temp:hptr;
BEGIN
  IF (p<>NIL) AND (root<>NIL) THEN
    BEGIN
      IF p=root THEN Delete:=DeleteMin(root,temp)
      ELSE
      BEGIN
        temp:=LinkSearch(p);
        IF temp=p THEN p^.parent^.left:=p^.parent^.left^.right
        ELSE temp^.right:=p^.right;
        Delete:=Merge(DeleteMin(p,temp),root);
      END;
    END ELSE Delete:=root;
END;
```

End Listings

Listing One *(Text begins on page 60.)*

```
/* ------------------------------------------------------------------------
    ERR_CODE.C Written by: William J. McMahon
    This module contains the functions used to manipulate error codes.
    Global Functions Defined Herein:
        err_combine(), err_format(), err_print()
------------------------------------------------------------------------ */
#include <stdio.h>
#include <limits.h>

#define ERR_BUMPER      10
#define ERR_THRESHOLD   (UINT_MAX/ERR_BUMPER)      /* ... for overflow. */

/* ----- Local Functions Defined Herein: ----- */
unsigned err_pop();
void     err_push();

#ifdef TEST             /* -------------- Test Harness ---------------- */

#define FIRST_ARG   0   /* Varies with compiler (0 or 1).               */
#define NCODES      32

main(argc, argv)
    int argc;
    char *argv[];
{
    unsigned err_combine();
    void     err_format();
    void     err_print();

    unsigned err_code;
    int      adder = 1;
    int      i;

    if (argc > FIRST_ARG)
        /* Override default starting code. */
        adder = atoi(argv[FIRST_ARG]);

    err_code = adder;
    printf("\nInput should be a mirror image of output.\n");
    printf("\n Input sequence: %d", err_code);
    for (i = 0; i < NCODES; ++i)        /* Build an error code, using   */
    {                                   /* multiple err_combine() calls.*/
        ++adder;
        if (adder >= ERR_BUMPER)
            adder = 1;
        printf("%d", adder);            /* Output RAW codes.            */
        err_code = err_combine(err_code, adder);
    }

    printf("\nOutput sequence: ");
    err_print(stdout, err_code);
}
#endif
/* ------------------------------------------------------------------------
    ERR_COMBINE Combines a new individual error code with an existing one.
    Returns: Combined error code.
------------------------------------------------------------------------ */
unsigned err_combine(
    unsigned original,      /* Original error code.                     */
    unsigned to_add)        /* Code to be added to it.                  */
{
    if ((original % ERR_BUMPER) == 0)   /* Some special codes are not   */
        return (original);              /* changed.                     */

    to_add %= ERR_BUMPER;               /* Make sure its in range.      */

    if (original > ERR_THRESHOLD)
    {   /* Prevent overflow. */
        err_push(original);
        original = 0;
    }

    return (original * ERR_BUMPER + to_add);
}

/* ------------------------------------------------------------------------
        ERR_FORMAT Decode and format an error code (and any overflow)
        into a string. Returns: Nothing.
------------------------------------------------------------------------ */
void err_format(
    char     *buffer,       /* Buffer to put formated code into.        */
    unsigned  err_code)     /* Error code to format.                    */
{
    char *p;
    p = buffer;
    while (err_code)
    {
        do
        {
            sprintf(buffer, "%d", err_code % ERR_BUMPER);
            buffer += strlen(buffer);
        }
        while ((err_code /= ERR_BUMPER) > 0);
        err_code = err_pop();
    }
}
/* ------------------------------------------------------------------------
        ERR_PRINT  Decode and output an error code (and any overflow).
    Returns: Nothing.
------------------------------------------------------------------------ */
void err_print(
    FILE     *stream,       /* Streem to output formated code to.       */
    unsigned  err_code)     /* Error code to output.                    */
{
    while (err_code)
    {
        do
        {
            fprintf(stream, "%d", err_code % ERR_BUMPER);
        }
        while ((err_code /= ERR_BUMPER) > 0);
        err_code = err_pop();
    }
}

/* ================ Local stack for overflow codes. ================== */
#define MAX_OVERFLOWS 10

static unsigned err_stack[MAX_OVERFLOWS];
static unsigned err_stack_top = 0;

/* ------------------------------------------------------------------------
ERR_POP  Returns: Combined error code of most recent overflow, 0 if none.
------------------------------------------------------------------------ */
static unsigned err_pop()
{
    if (err_stack_top <= 0)
        return (0);

    --err_stack_top;
    return (err_stack[err_stack_top]);
}

/* ------------------------------------------------------------------------
        ERR_PUSH  Push error code onto stack.
    Returns: Nothing.
------------------------------------------------------------------------ */
static void err_push(
    unsigned err_code)      /* Error code to save.                      */
{
    if (err_stack_top < MAX_OVERFLOWS)
    {
        err_stack[err_stack_top] = err_code;
        ++err_stack_top;
    }
}
```

End Listing

Listing One *(Text begins on page 66.)*

```
!SketchApp subclass of: Object !
className: SketchApp
superClass: Object
header:/* Class name: SketchApp */
/* Super name: Object */
/* File  name: Sketchpp */
/* header */
#include "objtypes.h"
extern id Object;
extern id Archiver;
extern id Notifier;
classVariables:/* class variables */instanceVariables:id objects;
id filename;     /* Filename string */
int changed;     /* changed flag */
fileName:Sketchpp !

!SketchApp class methods !
!SketchApp instance methods !

add_ drawObj
id drawObj;
{
    @self->objects add_ drawObj@;     /* add to collection */
    @self changed@;
}
!
asCollect
{
  return self->objects;
}
!
changed
{
  self->changed = TRUE;
}
!
clear
/* Erase the objects in our collection.  */
{
  extern id OrdCollect;
  if (self->filename) self->filename = @self->filename free@;
  if (self->objects) @self->objects freeAll@;
  self->objects = @OrdCollect new@;
  @self changed@;
  if (self->objects) return TRUE; else return FALSE;
}
!
getFname
{
  return self->filename;
}
!
loadFrom_ fname
id fname;
{
   id fstr, arc;

   if (fname)
   {
      arc = @Archiver openForLoad_ fname@;
      if (arc)
      {
         if (@self clear@) @self->objects freeAll@;
         self->objects = @arc getObject@;
         @self changed@;
         @arc free@;
         self->filename = fname;
         return TRUE;
      }
   }
   return FALSE;
}
!
main
{
}
main()
{
   id aView, aModel;
   extern id SketchView;
   aModel = @SketchApp new@;
   aView = @SketchView create_ "Sketch" ViewOf_ aModel@;
   if (aView)
   {
      @aView show@;
      @Notifier start@;
   }
}
!
save
{
   id arc;

   if (arc = @Archiver openForStore_ self->filename@)
    {
      @Notifier beginWait@;
      @arc putObject_ self->objects@;
      @arc free@;
      self->changed = FALSE;
      @Notifier endWait@;
    }
}
!
saveAs_ filename
id filename;
{
```

(continued on page 110)

Listing One *(Listing continued, text begins on page 66.)*

```
   if(self->filename) @self->filename free@;
   self->filename = filename;
   @self save@;
}
!
```

End Listing One

Listing Two

```
*** New PointArray methods:

getFrom_ anArchive
/* restore the receiver's size, recSize and size objects from the archive. */
id anArchive;     /* id of an Archiver object to store the receiver to */
{
   id anId;
   unsigned i;
   @super getFrom_ anArchive@;
   self->howMany = @anArchive getInt@;
}
putTo_ anArchive
/* Store the receiver's size, recSize and size objects to the archive. */
id anArchive;     /* id of an Archiver object to store the receiver to */
{
   id anId;
   unsigned i;
   @super putTo_ anArchive@;
   @anArchive putInt_ self->howMany@;
}
*** New Rect methods:
getFrom_ anArchiver
{
  self->left   = @anArchiver getInt@;
  self->top    = @anArchiver getInt@;
  self->right  = @anArchiver getInt@;
  self->bottom = @anArchiver getInt@;
}
putTo_ anArchiver
{
  @anArchiver putInt_ self->left@;
  @anArchiver putInt_ self->top@;
  @anArchiver putInt_ self->right@;
  @anArchiver putInt_ self->bottom@;
}
*** New PolyLine methods:
getFrom_ anArchiver
{
  self->ptArray = @anArchiver getObject@;
}
putTo_ anArchiver
{
  @anArchiver putObject_ self->ptArray@;
}
*** New Line methods:
getFrom_ anArchiver
{
  self->x0 = @anArchiver getInt@;
  self->y0 = @anArchiver getInt@;
  self->x1 = @anArchiver getInt@;
  self->y1 = @anArchiver getInt@;
}
putTo_ anArchiver
{
  @anArchiver putInt_ self->x0@;
  @anArchiver putInt_ self->y0@;
  @anArchiver putInt_ self->x1@;
  @anArchiver putInt_ self->y1@;
}
*** New Box methods:
getFrom_ anArchiver
{
  self->rec = @anArchiver getObject@;
}
putTo_ anArchiver
{
  @anArchiver putObject_ self->rec@;
}
*** New or changed SketchView methods:
* Add external ids for SketchApp, String, and FileSelect to SketchView class.
* Also remove instance variable "objects" and the main method.
*
erase
/* Erase the objects drawn on the receiver.  */
{
  if (self->model)
    {
     @self->model clear@;
     @self update@;
    }
}
initialize
/* Initialize receiver by creating a port to scribble on.   */
{
  id     aMenu;
  extern id    Port, Menu, PopupMenu;
  extern char  *scribNmes[];
  extern int   scribSels[], scribKeys[];
  if (aMenu = @Menu new@)
  {
  if (self->fMenu = @PopupMenu new_ "&Functions"@)
   {
```

```
    @self->fMenu setNames_ scribNmes sels_ scribSels keys_ scribKeys for_ self@;

    @aMenu append_ self->fMenu@;
    @self setMenu_ aMenu@;
    if (@self->model clear@)
     if (self->aPort = @Port createOn_ self@) /* make receiver graphics port */
       return self;
   }
  }
 return NIL;
}
char  *scribNmes[] = {"S&ketch\tF1", "&Line\tF2", "&Box\tF3", "E&rase\tF4",
                      "&Save\tF5", "Save&As\tF6", "&Open\tF7", NULL};

int   scribKeys[]  = { K_F1,          K_F2,        K_F3,       K_F4,
                       K_F5,          K_F6,        K_F7,       K_NULL};

int   scribSels[]  = {'sketch_',     'line_',     'box_',     'erase',
                      'save',       'saveAs',    'open', 0};

mouseDnX_ x Y_ y
/* Method to respond to mouse DOWN events.    */
int x,y;    /* position in receiver where mouse down event occured  */
{
  if (self->drawClass)
   {
    @Notifier captureMouseFor_ self@; /* capture all mouse events to window */
    @Notifier mouseTrkOn@;  /* enable mouse traking => mouse movement events */
    self->drawObj = @self->drawClass new@;  /* new instance of draw object */
    @self->model add_ self->drawObj@;     /* add to model */
    @self->drawObj setPort_ self->aPort@;
    return @self->drawObj mouseDnX_ x Y_ y@; /* pass forward mouse event */
   }
  else return FALSE;
}
open
{
   id fstr, str;
   fstr = @String newCStr_ "*.skh"@;
   str = @FileSelect ask_ "Select a Sketch file" for_ fstr of_ self@;
   if (str)
   {
      if (!!@self->model loadFrom_ str@) @str free@;
      @self update@;
   }
   @fstr free@;
}
paint
/* Draw all graphic objects contained by receiver in the objects ordered  */
/* collection.                                                            */
{
  int  sel = 'drawOn_';
  id retainedObjs;
  if (retainedObjs = @self->model asCollect@)
    @retainedObjs do_ sel with_ self->aPort@;
  return TRUE;
}
save
{
   if (@self->model getFname@) @self->model save@;
   else @self saveAs@;
}
saveAs
{
   id  fstr, str;
   fstr = @String newCStr_ "*.skh"@;
   str = @FileSelect ask_ "Select a Sketch file" for_ fstr of_ self@;
   if (str)
     {
      @self->model saveAs_ str@;
     }
   @fstr free@;
}
```

End Listings

Listing Four *(Text begins on page 76.)*

```
/* GPF386.C--for Phar Lap 386|DOS-Extender and Watcom C 386 7.0
wcl386 -DPROT_MODE -3r -mf -Ox -s gpf386
wdisasm gpf386 > gpf386.asm
run386 gpf386
NOTE!  To keep this example short, most of the precautions taken
in Listing 3 are not repeated here.  Refer to Listing 3 (GPFAULT.C)
for how the interrupt handler should really be written. */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <setjmp.h>
#include <dos.h>
#include "gpfault.h"

void reset_pharlap(void);
void prot_far *getvect_prot(short intno);
void real_far *getvect_real(short intno);
BOOL setvect_prot(short intno, void prot_far *handler);
BOOL setvect_real(short intno, void real_far *handler);
BOOL setvect(short intno, void prot_far *handler);
BOOL set2vect(short intno, void prot_far *phandler, void real_far *rhandler);
void revert(void);          // reinstall default handler
void interrupt far int13handler(REG_PARAMS r);
unsigned xtoi(char *s);

void prot_far *old_int13handler_prot;
void real_far *old_int13handler_real;
jmp_buf toplevel;
unsigned legal = 0;         // just a legal address to bang on
BOOL in_user_code = FALSE;

#define USE16           0x66
#define INT             0xcd
#define PUSH_DS         0x1e
#define POP_DS          0x1f
#define MOV_AX          USE16 0xb8
#define MOV_DS_ES       0x8c 0xc0 0x8e 0xd8      // MOV AX,ES/MOV DS,AX
#define XOR_AL          0x34
#define MOV_AX_CARRYFL  0x9c 0x58 XOR_AL 0x01   // PUSHF/POP AX/XOR AL,1
#define STI             0xfb

/* directives for compiler to generate inline code */
#pragma aux reset_pharlap = MOV_AX 0x01 0x25 /* 0x2501 */ INT 0x21 ;
#pragma aux getvect_prot = MOV_AX 0x02 0x25 /* 0x2502 */ INT 0x21 \
    parm [cl] value [ebx es] ;
 /*
    explanation of #pragma aux:  the preceding says that getvect_prot()
    takes one parameter in CL register, and returns value in ES:EBX.
    The "function" itself sets AX to 0x2502 and does an INT 0x21.
    When called: old_int13handler_prot = getvect_prot(0x0D);
    the compiler generates:
        mov cl, 0dh
        mov ax, 2502h
        int 21h
        mov old_int13handler_prot+4, es
        mov old_int13handler_prot, ebx
*/

#pragma aux getvect_real = MOV_AX 0x03 0x25 /* 0x2503 */ INT 0x21 \
    parm [cl] value [ebx] ;
#pragma aux setvect_real = \
    MOV_AX 0x05 0x25 /* 0x2505 */ \
    INT 0x21 \
    MOV_AX_CARRYFL \
    parm [cl] [ebx] value ;
#pragma aux setvect_prot = \
    PUSH_DS \
    MOV_DS_ES \
    MOV_AX 0x04 0x25 /* 0x2504 */ \
    INT 0x21 \
    POP_DS \
    MOV_AX_CARRYFL \
    parm [cl] [es edx] value ;
#pragma aux setvect = \
    PUSH_DS \
    MOV_DS_ES \
    MOV_AX 0x06 0x25 /* 0x2506 */ \
    INT 0x21 \
    POP_DS \
    MOV_AX_CARRYFL \
    parm [cl] [es edx] value ;
#pragma aux set2vect = \
    PUSH_DS \
    MOV_DS_ES \
    MOV_AX 0x07 0x25 /* 0x2507 */ \
    INT 0x21 \
    POP_DS \
    MOV_AX_CARRYFL \
    parm [cl] [es edx] [ebx] value ;

main()
{
    char buf[255];
    unsigned prot_far *fp;
    unsigned short seg;
    unsigned off, data;

    old_int13handler_real = getvect_real(0x0D);
    old_int13handler_prot = getvect_prot(0x0D);
    setvect(0x0D, (void prot_far *) int13handler);

    printf("'Q' to quit, '!' to reinstall default GP Fault handler\n");
    printf("%Fp is a legal address to poke\n", (void far *) &legal);
    printf("%Fp is not a legal address to poke\n", (void far *) (&legal-1));

    setjmp(toplevel);

    for (;;)
    {
        printf("$ ");
        *buf = '\0';
        gets(buf);
        if (toupper(*buf) == 'Q')
            break;
        else if (*buf == '!')
        {
            revert();
            continue;
        }
        // got bored of using sscanf()
        seg = xtoi(strtok(buf, ": \t"));
        off = xtoi(strtok(0, " \t"));
        data = xtoi(strtok(0, " \t"));
        in_user_code = TRUE;
        fp = MK_FP(seg, off);       // is this really user code?
        *fp = data;
        printf("poked %Fp with %x\n", fp, *fp);
        in_user_code = FALSE;
    }
    revert();
    printf("Bye\n");
    return 0;
}
void revert(void)
{
    if (! set2vect(0x0d, old_int13handler_prot, old_int13handler_real))
        printf("Can't revert!\n");
}
void interrupt far int13handler(REG_PARAMS r)
{
    _enable();
    reset_pharlap();
    if (in_user_code)
    {
        in_user_code = FALSE;
        printf("\nProtection violation at %04X:%08X\n",
            r.cs, r.ip);
        if (r.err_code)
            printf("Error code %04X\n", r.err_code);
        printf("<DS %04X> <ES %04X> <FS %04X> <GS %04X>\n",
            r.ds, r.es, r.fs, r.gs);
        printf("<EDI %08X> <ESI %08X> <FLAGS %08X>\n",
            r.di, r.si, r.flags);
        printf("<EAX %08X> <EBX %08X> <ECX %08X> <EDX %08X>\n",
            r.ax, r.bx, r.cx, r.dx);
        longjmp(toplevel, -1);
        /*NOTREACHED*/
    }
    else
    {
        printf("An internal error has occurred at %04X:%08X\n",
            r.cs, r.ip);
        revert();
        _chain_intr(old_int13handler_prot);
        /*NOTREACHED*/
    }
}
// convert ASCIIZ hex string to integer
unsigned xtoi(char *s)
{
    unsigned i =0, t;
    while (*s == ' ' || *s == '\t') s++;
    for (;;)
    {
        char c = *s;
        if      (c >= '0' && c <= '9') t = 48;
        else if (c >= 'A' && c <= 'F') t = 55;
        else if (c >= 'a' && c <= 'f') t = 87;
        else break;
        i = (i << 4) + (c - t);
        s++;
    }
    return i;
}
```

End Listing Four

Listing Five

```
/* GPF386.C--for Phar Lap 386|DOS-Extender and MetaWare High C for 386 MS-DOS
set ipath=\c386\inc\
\c386\hc386 gpfault -define PROT_MODE
\pharlap\fastlink gpfault -lib small\hce.lib
\pharlap\run386 gpfault
NOTE!  To keep this example short, most of the precautions taken
in Listing 3 are not repeated here.  Refer to Listing 3 (GPFAULT.C)
for how the interrupt handler should really be written.
*/

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <setjmp.h>
#include "msdos.cf"
#include "interrup.cf"
#include "gpfault.h"

BOOL call_pharlap(unsigned short ax, unsigned short cl);
void reset_pharlap(void);
IPROC getvect_prot(short intno);
void real_far *getvect_real(short intno);
BOOL setvect_prot(short intno, void prot_far *handler);
BOOL setvect_real(short intno, void real_far *handler);
BOOL setvect(short intno, IPROC handler);
BOOL set2vect(short intno, IPROC phandler, void real_far *rhandler);
void revert(void);  /* install old handlers */

#pragma Calling_convention(C_interrupt | _FAR_CALL);
void int13handler(REG_PARAMS r);
#pragma Calling_convention();
```

```
IPROC old_int13handler_prot;
void real_far *old_int13handler_real;
jmp_buf toplevel;
REG_PARAMS r2 = {0};
BOOL in_user_code = FALSE;

main()
{
    char buf[255];
    unsigned prot_far *fp;
    unsigned short seg;
    unsigned off, data;

    old_int13handler_real = getvect_real(0x0D);
    old_int13handler_prot = getvect_prot(0x0D);
    setvect(INT_GPFAULT, int13handler);

    printf("'Q' to quit, '!' to reinstall default GP Fault handler\n");

    if (setjmp(toplevel) == -1)
    {
        printf("Protection violation at %04X:%08X\n",
            r2.cs, r2.ip);
        if (r2.err_code)
            printf("Error code %04X\n", r2.err_code);
        printf("<ES %04X> <DS %04X> <EDI %08X> <ESI %08X> <FLAGS %08X>\n",
            r2.es, r2.ds, r2.di, r2.si, r2.flags);
        printf("<EAX %08X> <EBX %08X> <ECX %08X> <EDX %08X>\n",
            r2.ax, r2.bx, r2.cx, r2.dx);
    }
    for (;;)
    {
        printf("$ ");
        *buf = '\0';
        gets(buf);

        if (toupper(*buf) == 'Q')
            break;
        else if (*buf == '!')
        {
            revert();
            continue;
        }
        sscanf(buf, "%04X:%08X %x", &seg, &off, &data);
        FP_SEG(fp) = seg;
        FP_OFF(fp) = off;
        in_user_code = TRUE;
        *fp = data;
        printf("poked %p with %x\n", fp, *fp);
        in_user_code = FALSE;
    }
    revert();
    printf("Bye\n");
    return 0;
}
void revert(void)
{
    set2vect(0x0d, old_int13handler_prot, old_int13handler_real);
}
#pragma Calling_convention(C_interrupt | _FAR_CALL);
void int13handler(REG_PARAMS r)
{
    if (in_user_code)
    {
        in_user_code = FALSE;
        r2 = r;
        reset_pharlap();
        STI;    // _inline(0xFB):  reenable interrupts
        longjmp(toplevel, -1);
    }
    else
    {
        printf("Internal error at %04X:%08X\n", r.cs, r.ip);
        revert();
        return;     // let the fault happen again  (no _chain_intr)
    }
    /*NOTREACHED*/
}
#pragma Calling_convention();
BOOL call_pharlap(unsigned short ax, unsigned short cl)
{
    Registers.AX.W = ax;
    Registers.CX.LH.L = cl;
    calldos();
    return !(Registers.Flags & 1);
}
void reset_pharlap(void)
{
    call_pharlap(0x2501, 0);
}
IPROC getvect_prot(short intno)
{
    IPROC handler;
    call_pharlap(0x2502, intno);
    /* no MK_FP for High C 386 */
    FP_SEG(handler) = Registers.ES.W;
    FP_OFF(handler) = Registers.BX.R;
    return handler;
}
void real_far *getvect_real(short intno)
{
    call_pharlap(0x2503, intno);
    return (void real_far *) Registers.BX.R;
}
BOOL setvect_prot(short intno, void prot_far *handler)
{
    Registers.DS.W = FP_SEG(handler);
    Registers.DX.R = FP_OFF(handler);
    return call_pharlap(0x2504, intno);
}
```

(continued on page 114)

Listing Five *(Listing continued, text begins on page 76.)*

```
BOOL setvect_real(short intno, void real_far *handler)
{
    Registers.BX.R = (unsigned) handler;
    return call_pharlap(0x2505, intno);
}
BOOL setvect(short intno, IPROC handler)
{
    Registers.DS.W = FP_SEG(handler);
    Registers.DX.R = FP_OFF(handler);
    return call_pharlap(0x2506, intno);
}
BOOL set2vect(short intno, IPROC phandler, void real_far *rhandler)
{
    Registers.DS.W = FP_SEG(phandler);
    Registers.DX.R = FP_OFF(phandler);
    Registers.BX.R = (unsigned) rhandler;
    return call_pharlap(0x2507, intno);
}
```

End Listing Five

Listing Six

```
/* OS2TRACE.C--catching GP faults in OS/2, using DosPTrace()
compile with:
    cl -Lp os2trace.c
to make tiny (less than 3K) .EXE with C run-time DLL, compile with:
    cl -AL -c -Gs2 -Ox -Lp -I\msc\inc\mt os2trace.c
    link /nod/noi crtexe.obj os2trace,os2trace,,crtlib.lib \os2\lib\os2.lib;
*/

#include <stdio.h>
#include <string.h>
#include <process.h>
#include <signal.h>
#include <setjmp.h>

#define INCL_DOS
#define INCL_DOSTRACE
#define INCL_VIO
#include "os2.h"
#include "ptrace.h"

#define LOCAL static
typedef unsigned WORD;

LOCAL VOID NEAR print_regs(void);
LOCAL VOID NEAR send_sig_segv(void);
LOCAL VOID NEAR catch_sig_segv(void);
LOCAL WORD NEAR trace(int argc, char *argv[], BOOL verbose);
LOCAL VOID NEAR start_trace(char *prog, char *cmdline);
LOCAL char *progname(char *s);
LOCAL char *cmdline(int argc, char *argv[]);
LOCAL WORD NEAR selector_from_segment(WORD seg);
LOCAL VOID NEAR set_csip(WORD cs, WORD ip);
LOCAL int break_handler(void);

PTRACEBUF ptb;
jmp_buf toplevel;

#define FP_OFF(fp)      ((unsigned) (fp))

#define JMP_GPFAULT     -1
#define JMP_BREAK       -2

main(int argc, char *argv[])
{
    char buf[80];
    BOOL do_trace = TRUE;
    BOOL verbose = FALSE;
    WORD x;
    int i;
    for (i=1; i<argc; i++)
        if (argv[i][0] == '-')
            switch(argv[i][1])
            {
                case 'x': do_trace = FALSE; break;
                case 'v': verbose = TRUE;   break;
            }
    signal(SIGINT, break_handler);  // doesn't work with CRTLIB.DLL
    if (do_trace)
        return trace(argc, argv, verbose);
    // if (! do_trace) run interpreter
    switch (setjmp(toplevel))   // used to catch multiple events
    {
        case JMP_GPFAULT:
            printf("General Protection violation!\n");
            break;
        case JMP_BREAK:
            printf("break\n");
            signal(SIGINT, break_handler);
            // what if this is trace process??
            break;
    }
    for (;;)
    {
        printf("$ ");
        gets(buf);
        if (toupper(*buf) == 'Q')
            break;
        // cause one of a number of different GP faults
        switch (*buf)
        {
            case '0': x = *((int far *) 0L);        break;  // GP fault
            case '1': x = *((int far *) -1L);       break;  // GP fault
            case '2': *((char *) main) = 'x';       break;  // bashes data!
            case '3': x = VioWrtTTY(0L, 100, 0);    break;  // GP fault in DLL
            case '4': x = VioWrtTTY(-1L, 100, 0);   break;  // GP fault in DLL
            case '5': x = VioWrtTTY(0L);            break;  // boom!
            case '6': x = puts(-1L);                break;  // GP fault
            case '7': x = DosGetInfoSeg(1L, 2L);    break;  // thread dies
        }
        printf("Executed statement %c\n", *buf);
    }
    return 0;
}
/*  case 2 is important because, though the operation is illegal, it does
    not generate a GP fault.  Consequently, the operation is successful.
    Depending on how OS2TRACE.C is compiled, sometimes a string gets bashed
    so that string prints out "General protection vixlation," sometimes
    something else gets bashed so that when we exit, C run-time puts up a
    message about null pointer assignment.
*/
LOCAL WORD NEAR trace(int argc, char *argv[], BOOL verbose)
{
    start_trace(progname(argv[0]), cmdline(argc, argv));
    /* DosPTrace event loop */
    for (;;)
    {
        ptb.tid = 0;
        ptb.cmd = GO;
        DosPTrace(&ptb);
        switch (ptb.cmd)
        {
            case EVENT_GP_FAULT:
               if (verbose)
               {
                   printf("GP fault (error %04X) at %04X:%04X\n",
                        ptb.value, ptb.segv, ptb.offv);
                   print_regs();
               }
               send_sig_segv();
               break;
           case EVENT_THREAD_DEAD:
               if (verbose)
                   printf("Thread %u dying\n", ptb.tid);
               break;
           case EVENT_DYING:
               if (verbose)
                   printf("Process %u dying\n", ptb.pid);
               return 0;
        }
    }
    /*NOTREACHED*/
}
LOCAL VOID NEAR print_regs(void)
{
     ptb.cmd = READ_REGISTERS;
     DosPTrace(&ptb);
     printf("AX=%04X BX=%04X CX=%04X DX=%04X SI=%04X DI=%04X BP=%04X\n",
         ptb.rAX, ptb.rBX, ptb.rCX, ptb.rDX, ptb.rSI, ptb.rDI, ptb.rBP);
     printf("DS=%04X ES=%04X IP=%04X CS=%04X FL=%04X SP=%04X SS=%04X\n",
         ptb.rDS, ptb.rES, ptb.rIP, ptb.rCS, ptb.rF,  ptb.rSP, ptb.rSS);
}
LOCAL VOID NEAR send_sig_segv(void)
{
    /* because of OS/2 "disjoint LDT space," we could just as easily say:
        WORD cs = FP_SEG((void far *) catch_sig_segv);
    it will be mapped to same selector in both processes
    */
    WORD cs = selector_from_segment(1);     // catch_sig_segv() in seg 1
    WORD ip = FP_OFF((void far *) catch_sig_segv);
    set_csip(cs, ip);
}
LOCAL WORD NEAR selector_from_segment(WORD seg)
{
    ptb.value = seg;
    ptb.cmd = SEG_NUM_TO_SELECTOR;
    DosPTrace(&ptb);
    return (ptb.cmd == EVENT_ERROR) ? 0 : ptb.value;
}
LOCAL VOID NEAR set_csip(WORD cs, WORD ip)
{
    ptb.cmd = READ_REGISTERS;
    DosPTrace(&ptb);
    ptb.rCS = cs;
    ptb.rIP = ip;
    if (ptb.rDS != ptb.rSS)
    {
        printf("Faulted inside DLL code\n");
        ptb.rDS = ptb.rSS;  // very important!
    }
    ptb.cmd = WRITE_REGISTERS;
    DosPTrace(&ptb);
}
LOCAL VOID NEAR catch_sig_segv(void)
```

```
{
    longjmp(toplevel, JMP_GPFAULT);
}
// shared by debugger and debuggee processes
LOCAL int break_handler(void)
{
    signal(SIGINT, SIG_IGN);
    longjmp(toplevel, JMP_BREAK);
}
LOCAL VOID NEAR start_trace(char *prog, char *cmdline)
{
    RESULTCODES resc;
    if (DosExecPgm(NULL, 0, EXEC_TRACE, cmdline, NULL, &resc, prog) != 0)
        return;
    ptb.pid = resc.codeTerminate;
    ptb.tid = 0;
}
// tacks .EXE after program name
LOCAL char *progname(char *s)
{
    static char str[128];
    strcpy(str, s);
    strcat(str, ".EXE");
    return str;
}
// undoes all argc/argv work, appends -x switch
LOCAL char *cmdline(int argc, char *argv[])
{
    static char str[128], *t = str;
    char *s = argv[0];
    register int arg = 0;
    while (arg < argc)
    {
        while (*s)
            *t++ = *s++;
        *t++ = (arg) ? ' ' : '\0';  // '\0' after program name
        s = argv[++arg];
    }
    *t++ = '-'; *t++ = 'x';         // append -x switch
    *t = '\0';
    return str;
}
```

End Listing Six

Listing Seven

```
// ptrace.h

// DosPTrace commands
#define READ_I_SPACE            0x0001
#define READ_D_SPACE            0x0002
#define READ_REGISTERS          0x0003
#define WRITE_I_SPACE           0x0004
#define WRITE_D_SPACE           0x0005
#define WRITE_REGISTERS         0x0006
#define GO                      0x0007
#define TERMINATE_CHILD         0x0008
#define SINGLE_STEP             0x0009
#define STOP_CHILD              0x000A
#define FREEZE_CHILD            0x000B
#define RESUME_CHILD            0x000C
#define SEG_NUM_TO_SELECTOR     0x000D
#define GET_FLOATINGPT_REGS     0x000E
#define SET_FLOATINGPT_REGS     0x000F
#define GET_DLL_NAME            0x0010
#define THREAD_STATUS           0x0011   // new
#define MAP_READONLY_ALIAS      0x0012
#define MAP_READWRITE_ALIAS     0x0013
#define UNMAP_ALIAS             0x0014

// DosPTrace events
#define EVENT_SUCCESS           0
#define EVENT_ERROR             -1
#define EVENT_SIGNAL            -2
#define EVENT_SINGLESTEP        -3
#define EVENT_BREAKPOINT        -4
#define EVENT_PARITYERROR       -5
#define EVENT_DYING             -6
#define EVENT_GP_FAULT          -7
#define EVENT_LOAD_DLL          -8
#define EVENT_FLOATPT_ERROR     -9
#define EVENT_THREAD_DEAD       -10
#define EVENT_ASYNC_STOP        -11
#define EVENT_NEW_PROCESS       -12
#define EVENT_ALIAS_FREE        -13

// DosPTrace error types
#define ERROR_BAD_COMMAND       1
#define ERROR_CHILD_NOTFOUND    2
#define ERROR_UNTRACEABLE       5

// Thread states
#define THREAD_RUNNABLE         0
#define THREAD_SUSPENDED        1
#define THREAD_BLOCKED          2
#define THREAD_CRITSEC          3

// Thread debug states
#define THREAD_THAWED           0
#define THREAD_FROZEN           1
```

End Listings

Programming RISC Engines

Instruction sets are designed for fast, pipelined execution

Neal Margulis

The innovation of the assembly line revolutionized manufacturing. By breaking assembly down into simple steps, tremendous efficiency is gained. Instead of each unit taking one hour to produce, it can be broken down into six steps of ten minutes each. This results in a new unit produced every ten minutes without increasing the manufacturing machinery needed. As long as each step takes about the same time, the assembly line is kept filled and the throughput is much higher than if all operations are done serially.

RISC processors rely on the same concept. They define an instruction set that can efficiently move through the processor's pipeline. To do this, the instruction set must meet several criteria. First, each instruction takes only one clock to execute. Also, each instruction must be easily fetched and quickly identified. Unlike their CISC counterparts, RISC processors expose some of their pipeline to software and allow the programmer to arrange instruction sequences to avoid pipeline freezes.

Compilers are an important part of developing code for RISC processors. RISC instructions are highly regular in form and have consistent behavior, allowing compilers to generate efficient code. However, the opportunities to further tune portions of a program by hand-coded assembly language always exist. Also, hardware-specific portions of device drivers are convenient to write in assembly language. This article shows how RISC instructions operate and gives some examples including C compiler-generated machine code.

The Intel i860 microprocessor exemplifies a modern RISC processor. It includes other features such as floating point, memory management, and caches on one chip, but at its heart is an efficient RISC core that, like other RISC processors, is easy to program. The i860's RISC core architecture will be used for the examples in this article, but the concepts discussed extend to most other RISC processors as well. With a general understanding of how the RISC core's pipeline is organized, you will gain insight into how to order instruction sequences.

Neal is chief applications engineer for high-performance processors at Intel Corp. and can be reached at 2625 Walsh Ave., SC4- 40, Santa Clara, CA 95051. Neal is the author of the i860 Programmer's Guide, *(Osborne/McGraw-Hill) due out this spring.*

Programming Goals

The technique for programming RISC processors involves understanding the instructions and, equally important, the interaction between instructions. The acronym "reduced instruction set computer" (RISC) alerts you that some instructions are no longer available. While this may be true, they have been replaced with a powerful set of simple operations. Together, these simple operations perform the same functions as the more complex instructions you are used to. The RISC programming challenge is to sequence these operations to get maximum performance from the processor.

If each instruction takes only one clock at each pipeline stage, then the number of instructions executed per second is equal to the processor's clock frequency. After adjusting to allow for uncooperative instructions, a measure that is often called the "native MIPS" rating can be calculated as follows:

$$\textit{Native MIPS} = \frac{\textit{Clock Frequency}}{\textit{Clocks per Instruction}}$$

This number has little meaning. The native MIPS rating has been compared to RPMs (revolutions per minute) of a car: It tells you how fast the engine is going, but not how fast the car is going. Instead, the native MIPS rating needs to be normalized to a common metric that indicates what you really want to measure — the time per task. Such as miles per hour for the car, the VAX MIPS rating has become the standard. The most common method for calculating VAX MIPS is through benchmarking: Compare the duration of a task on the processor with its duration on the VAX 11/780. The task's time ratio is the VAX MIPS rating. Do not attempt to gain information on a processor's native MIPS rating from its VAX MIPS rating.

The time per task can also be calculated analytically as follows:

*time/task= (instructions/task) * (clocks/instruction) * (time/clock).*

While RISC processors may require more instructions per task than traditional processors, they more than make up for it by reducing the average clocks per instruction and increasing the clock speed.

Instruction Processing

In order to execute each instruction efficiently, processing is broken into four stages. Each stage performs a designated operation in one cycle and passes the instruction to the next stage. The stages are "Fetch," "Decode," "Execute," and "Write."

The Fetch stage gets the instruction from the instruction cache into an internal storage latch. The Decode stage

accesses the source registers and decodes the instruction. The ALU operation is performed in the Execute stage. Address calculation is done here for memory operations. The results of the instruction are written to the register file in the Write stage if the instruction was not a memory operation. The data cache access is accessed here for memory operations.

To allow each stage of the pipeline to complete in one cycle, careful attention is paid to the instruction format. Figure 1 shows the general format for all of the RISC core instructions. All instructions are 32 bits long with designated fields for the opcodes and registers. To access any of the i860's 32-integer registers, 5 bits are used for each register designator. Without having to decode the length of the instruction, or take multiple cycles to read the instruction into the processor, the Fetch stage can execute in one cycle. In the Decode stage, the source register accesses can begin before the instruction type is known, because the field within the instruction that indicates the source register is always in the same place. By allowing only instructions that can be executed in one stage, the Execute stage is always performed in one cycle. During the Write stage, the result is written back into the register file.

Besides allowing efficient instruction execution, the four-stage pipeline allows instructions to be overlapped. Overlapping the instructions allows a new instruction to start with each clock cycle. Figure 2 illustrates the resulting speed-up of overlapping instructions. With sequential (scalar) execution, each instruction passes through all four stages of the pipeline before the next instruction starts. With pipelined execution, instructions start as soon as the previous instruction enters the second stage. In the same number of cycles that sequential execution processes three instructions, pipelined execution processes 12 instructions, a fourfold improvement.

To maintain performance, the pipeline needs to keep all of the stages active all of the time. If an instruction were to take two cycles in any of the stages, it would cause the other three stages to wait an extra cycle. This is referred to as a freeze condition. Although all instructions are designed to take only one cycle in each stage, freeze conditions can occur. The two types of instructions that are most likely to cause freezes in the pipeline are memory operations and branch instructions. RISC processors define the instructions to allow the programmer to reduce the occurrence of such freeze.

Load/Store Instructions

Unlike earlier processors that allow operations on data in memory, the only memory operations permitted on RISC processors are loads and stores. All other operations are performed directly on the values in the registers. The load/store architecture simplifies the design of the processor and allows the programmer to hide the delay caused by memory accesses.

Loads from memory always have at least a one-clock delay, even if the data is in the on-board cache. Figure 3 shows a pipeline sequence for a load instruction and the subsequent two instructions. The data from the load operation is available at the end of the load instruction's Write stage. This is too late for the instruction immediately following the load to use the data as a source operand. The instruction slot following a load is called the "load-delay slot."

The i860 gives the programmer two options for the load-delay slot. The

(continued on page 128)

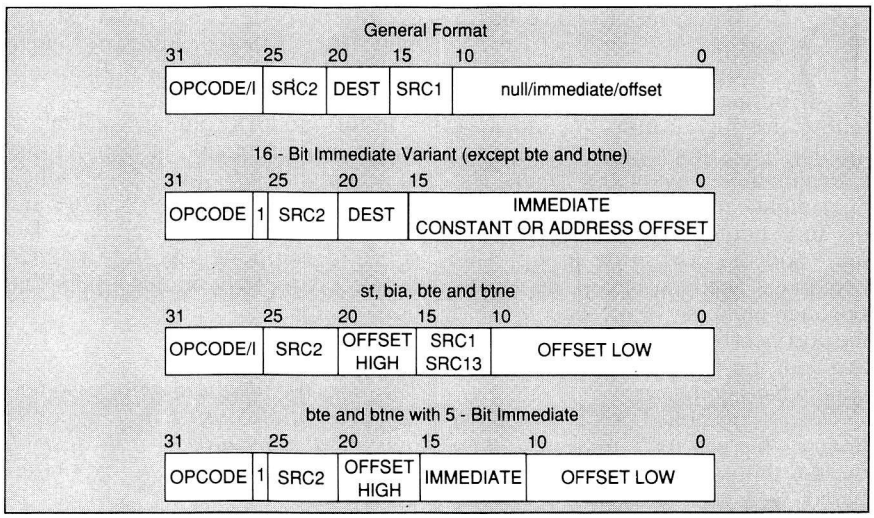


Figure 1: *The general format for all of the RISC core instructions*

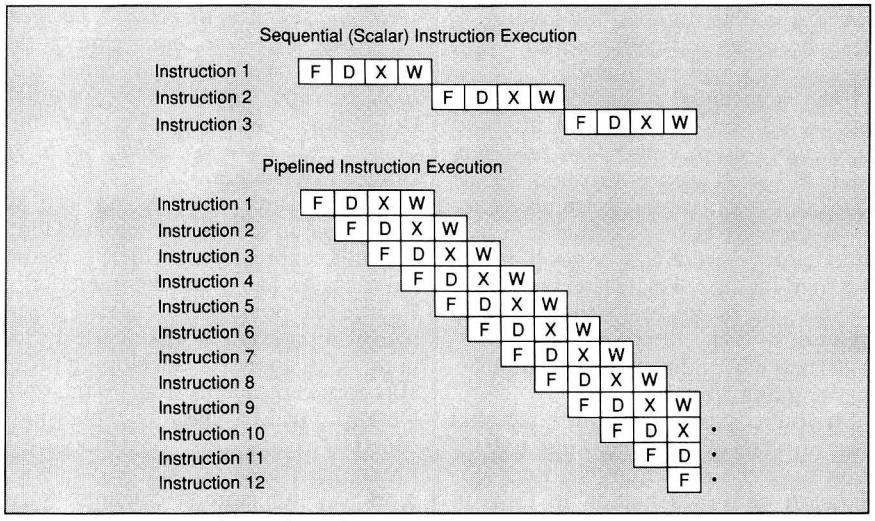


Figure 2: *The speed-up of overlapping instructions*

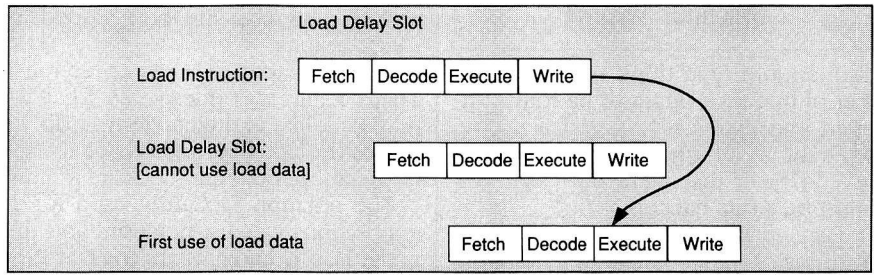
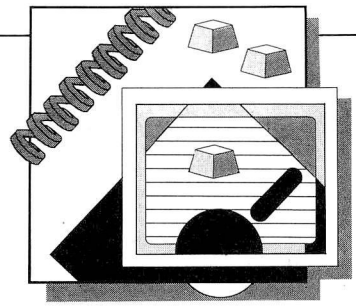


Figure 3: *A pipeline sequence for a load instruction and the subsequent two instructions*



Four Hundred and Eighty-seven

Michael Swaine

DON'T stop me if you've heard this one.

A man is sent to prison, and during his first supper inside, a fellow prisoner stands up and says loudly, "Five thousand nine hundred and thirty-three." Everybody laughs. The next night, a different convict jumps up and shouts, "Three thousand and ten," and the crowd breaks up. This continues: The next day in the yard, a prisoner mutters, "Two sixteen," and the prisoners near him snicker and snort. Puzzled, the man finally asks his cell mate what's going on.

"There's one joke book in the prison library," his cell mate explains, "with the ten thousand best jokes, all numbered. Well, a lot of us have been here long enough to have memorized the jokes, so when we want to tell one, we just use its number. My favorite is four hundred and eighty-seven." And he chuckles softly to himself.

The next night at supper, the man stands up and shouts, "Four hundred and eighty-seven." His fellow prisoners stare at him in silence. Crushed, he sits back down and tries to finish his meal.

"I thought you said four hundred and eighty-seven was a good joke," he snarls at his cell mate later that night.

"It's one of the best," the cell mate drawls.

"Then why didn't anybody laugh?"

"You didn't tell it right."

If you're wondering right now what the number of the joke I just told is, you have the right mind-set for a discussion of representation in Lisp. If, furthermore, you think that the number of that joke ought to be four hundred and eighty-seven, and if you've thought about what a different joke it would be if that were the case, you must be a Lisp hacker.

Lisp hackers often point to certain features of the language in explaining why they choose to program in Lisp. These include recursion, the ability to operate on code as data, and extensibility. None of these features are unique to Lisp, though, and none of them gets across why Lisp is unique.

In this month's column, we'll take a look at representation in Lisp, and try to get some sense of the power inherent in Lisp's representation scheme, and we'll examine the wide range of data structures supported by the Common Lisp standard, built on this representation scheme.

Notes on Notation

"Author: The idea is to imitate Gödel's self-referential construction, which as you know is INDIRECT, and depends on the isomorphism set up by Gödel-numbering.

"Crab: Oh. Well, in the programming language Lisp, you can talk about your own program directly . . . because programs and data have exactly the same form. Gödel should have just thought up Lisp. . . ." — Douglas Hofstadter, a fictional character, and Crab, a crab, *in Gödel, Esher, Bach,* by Douglas Hofstadter.

John McCarthy thought up Lisp 30 years ago as a tool for manipulating symbolic expressions, which is essential for tasks like symbolic integration. But the real point was to make it possible for a program to talk about a program, with an eye to developing provably correct programs. Toward this end, McCarthy used a class of expressions called "s-expressions," or symbolic expressions, based on Alonso Church's work on the lambda calculus. S-expressions permit programs and data in Lisp to have the same form. Programs and data in Lisp are both coded as s-expressions.

What, exactly, are s-expressions? That's really two questions: What do they look like and what do they do, or what's the notation and what's the interpretation? Both questions are fruitful.

The notation McCarthy used for s-expressions was list notation, and the name Lisp is an acronym for LISt Processing. Roughly, s-expressions are lists. But the matter of s-expression notation goes a little deeper than this.

Formally, an s-expression is either an atomic symbol (atom) or a list of s-expressions:

```
<s-expr> ::= <atom> ¦
                <list-of-s-expressions>
```

What's an atomic symbol? An atomic symbol, or atom, is, formally, just a string of characters subject to certain constraints. It's a name, a symbol, such as a name for a variable or constant or function in any language. Or, as we shall see, not quite like a name in any other language.

And the notation for lists? A list can be recursively defined to be either the empty list (which is sometimes written using the atom NIL) or an item like the thing on the right below:

```
<list> ::= NIL ¦ ( <s-expr> <list> )
```

In other words, all Lisp lists are binary trees with all their (non-NIL) atoms hanging off their left terminal branches, and all right terminal branches containing NILs. NIL is a primitive Lisp object (atom) used for several purposes, including to represent Boolean false and the empty list.

An alternative notation for lists used in Lisp is called "dot" notation. In dot notation, a (dotted) list can be defined thus:

```
<dotted-list> ::= ( <s-expr>. <s-expr> )
```

Some examples of Lisp s-expressions are:

```
A
(A . B)
(A . NIL)
(A . (B . NIL))
(A . (B . ((C . (D . NIL)) . NIL)))
```

Here are prose descriptions of the five s-expressions above:

A is not a list, nor even a dotted list, but is both an atomic symbol and an s-expression.

(A . B) is a dotted list, but not a true list, because it has a non-NIL right terminal branch.

(A . NIL) is the list containing one element: The atomic symbol A.

(A . (B . NIL)) is the list containing the two elements A and B.

(A . (B . ((C . (D . NIL)) . NIL))) is the list containing three elements: A, B, and the list containing the two elements C and D.

Figure 1 shows pictures of the binary trees that the s-expressions represent. There are reasons for using dot notation in low-level programming, but for most purposes the simpler list notation is used. In this shorthand form, the above forms are written:

(No list representation for A, since A is not a list.)
(No list representation for (A . B), since (A . B) is not a "true" list.)
(A)
(A B)
(A B (C D))

Lists in this notation can have any number of elements, but note that the underlying representation is still the binary tree described by the dotted pairs of the corresponding dot notation. This (non-dotted) notation is the form in which all Lisp programs are normally written. Every Lisp program or subprogram begins and ends with a matched pair of parentheses, usually with more pairs inside and with atomic symbols inside them. Every data object is also expressed as one of these parenthesized lists if it is not represented as an atomic symbol.

Lisp students early on decided that the acronym Lisp stood for "Lots of Insipid, Stupid Parentheses." Little wonder.

How Symbols Symbolize

"Author: I find indirect self-reference a more general concept, and far more stimulating, than direct self-reference. Moreover, no reference is truly direct — every reference depends on SOME kind of coding scheme. It's just a question of how implicit it is. Therefore, no self-reference is direct, not even in Lisp."
— ibid.

We've looked at the notation used to write Lisp programs (list notation) and at the more elemental dot notation and the underlying data structure that lists form. We've seen that lists contain other lists and/or atomic symbols, and that properly-formed combinations of lists and atomic symbols are called symbolic expressions, or s-expressions, and that s-expressions are the universal form in which data objects and programs in Lisp are expressed. So far, these are just notational issues, and don't get at the power of Lisp representation. Now let's consider what these s-expressions mean.

Atomic symbols are named data objects. The object named can be a constant or a variable, and the structure of the object can be anything. Each symbol has associated with it a structure called its "property list," or p-list, which allows it to be treated as a record structure with an extensible set of components, each of which can be an arbitrarily complex named data object. The name can be retrieved, given the object, or can be used to retrieve the data object, and there are functions for creating and deleting data objects, and for manipulating their property lists. The tools for manipulating symbols are powerful, but the real power of Lisp comes in the ability to treat tools themselves symbolically, and operate on them. A Lisp named function is just one particular kind of named data object that atomic symbols can comprise.

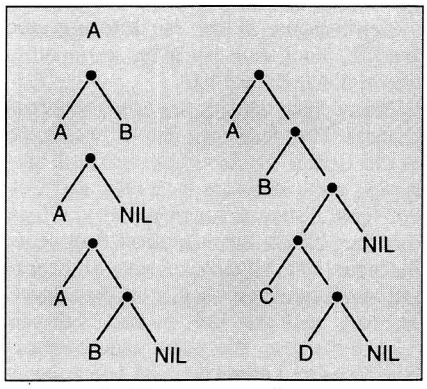


Figure 1: *Binary trees that s-expressions represent*

The difference between constants and variables is roughly one scope of binding. In Lisp, the values bound to constants and variables are maintained in different data structures.

A Lisp constant is an atomic symbol that possesses an associated constant value. One of the things that can be stored in the p-list of an atomic symbol is what is called an "apval" (constant value). If a symbol has the string "apval" as the first element of a pair of values on its p-list, the symbol is a constant, and its constant value is the second element of that pair.

Variables in Lisp are atomic symbols, too, but they don't have apvals. They get their values via a different list. Lisp maintains a data structure called the "a-list," or association list. This is a list of symbol value pairs representing the

current bindings of variables and of function names. At any point while Lisp is running, we are "inside" an s-expression; that is, an s-expression is being evaluated. The function that does the evaluation, called "eval," maintains the a-list. Constant values are not affected by a-list bindings, and the p-list is checked before the a-list when retrieving a value for a symbol.

Named functions are also maintained by eval, and the binding stored on the a-list. It is worth noting that functions are Lisp objects, and have a data type. All functions are of type *function*.

The meaning of a list is generally dependent on the meanings of its constituent atomic symbols, because a list is a data structure. One use of lists, though, attaches quite a different sort of meaning to a list. This is the use of a list as the invocation of a function. The fundamental unit of interaction with Common Lisp is the form, which is an s-expression, ergo a data object. The form is handed to eval to be interpreted as a function, with a new data object to be returned. In this interpretation, the first element of the list names a function to be applied to the other elements as arguments. (This is oversimplified.)

Although the use of a list to represent the invocation of a function and its use to represent a particular data object may seem very different, and are in truth very different, the style of Lisp programming often makes the two uses seem very much alike. Consider the function *FIRST*, which takes a list as an argument and returns the first element of the list as its value. When you write

```
(FIRST VENDOR_LIST)
```

you are executing a function on the list *VENDORLIST*. But it's obvious that you are also referring to a particular data object, specifically the *FIRST* of *VENDOR_LIST*. To the implementer of the Lisp system, it matters that *FIRST* is a function, but for most purposes, the user can think of it as a way of referring to a particular element of a complex data structure. This is a property of functions generally, not restricted to Lisp, but the complete dependence of Lisp on functions makes this a distinctive aspect of the "feel" of programming in Lisp.

Common Lisp Data Types

In Common Lisp, an object may belong to more than one data type, so it is not generally meaningful to inquire about *the* data type of an object. For example: the simplest data type is type null. It consists of exactly one item: NIL. NIL is both list and atom, it means both false and the empty list, and it can be written NIL or ( ). It is possible to inquire whether an object belongs to a particular type, and a Boolean function *typep* is provided for this purpose. It is objects, however, and not variables, that can be typed. A Lisp variable can have any Lisp object as its value.

Dotted lists, true lists, and trees are all different ways of viewing structures of conses. A cons is a Lisp data type, and is another name for the dotted pair: (< s-expr > . < s-expr >). What we have been calling a list is a chain of conses branching off the right-side s-expressions. If the list is terminated by NIL, it's a true list, if by some other atom, it's a dotted list.

Many Lisp data types are built from conses. The data type list, for example, is the union of the cons and null data types, so it includes true lists and dotted lists. Lists are supported by a large number of list-manipulation functions. Lists can also be used as sets, and there are set functions, such as union, intersection, and the like, in the Common Lisp definition. The a-list, used by function eval to keep track of bindings, is a list of conses.

Common Lisp also supports sequences and other compound data types, hash tables, arrays, vectors, strings, streams, and structures, among other data types. Conceptually, these, too, can be viewed as being built of cons cells, although some of them may be implemented in some more efficient fashion. Type sequence includes both lists and vectors, which are one-dimensional arrays. An array is an object with components arranged according to a Cartesian coordinate system. Arrays can have their sizes adjusted dynamically after creation, or can be of type simple array, and not permit this. Non-simple arrays can also share data with other non-simple arrays. Element *1,1* of a two-dimensional array *foo* is specified via the array-reference function *aref* thus: (*aref foo 1 1*). The data type vector is a subtype of type array. A string is a vector of type *string-char*. A structure is an instance of a user-defined data type with a fixed number of named components. These are similar to Pascal records.

Some Lisp data objects are what is called "self-evaluating forms." Because, as I suggested earlier, Lisp is always evaluating forms, even objects such as numbers, characters, and the empty list get evaluated. These objects return themselves as values. Common Lisp defines four broad kinds of numeric data types: integer, ratio, floating point, and complex; and one-character data type, with subtypes.

Integers can be of any length, and are implemented by means of two subtypes: fixnums and bignums. Fixnums are limited in length and are more efficient to use; bignums can be any length. The definition of Common Lisp calls for the distinction between fixnums and bignums to be hidden from the user most of the time. The default radix is decimal, but any radix from 2 to 36 can be specified.

The ratio data type permits precise representation of rational numbers, such as the fraction 2/3. Common Lisp always converts a ratio to lowest terms, and to an integer if possible.

Common Lisp provides names but not full specifications for four different floating-point data types. These are short, single, double, and long.

Complex numbers are represented in Cartesian form as pairs of numbers of any of the other data types; thus the actual type of a complex number is the type complex and the type of the components, such as complex short float. The components are coerced to have the same type.

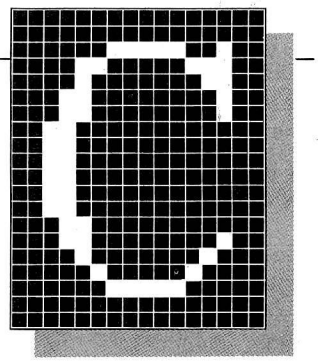
There are two important Common Lisp character data subtypes, *standard-char* and *string-char*. The former is intended to be as portable a character set as possible; that is, programs written in it should port easily. This is important, because the character-data type provides for such features as specification of font attributes and various modifying flags, the latter for programmers with terminals from Mars. The *string-char* subtype comprises all characters that can be contained in strings. These can't have any of those funny flags or font attributes attached to them.

Next month we'll look at the full hierarchy — although one author has called it a heterarchy — of Common Lisp data types, and at some of the object-oriented extensions to Common Lisp. We'll also look at the way in which Lisp functions are interpreted, which will get us into what Douglas Hofstadter has called a double-entendre:

> . . double-entendre can happen with Lisp programs that are designed to reach in and change their own structure. If you look at them on the Lisp level, you will say that they change themselves; but if you shift levels, and think of Lisp programs as data to the Lisp interpreter . . . then in fact the sole program that is running is the interpreter, and the changes being made are merely changes in pieces of data."

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 9.**



TEXTSRCH Connections

Al Stevens

This month we continues with the TEXTSRCH project, one that started two months ago. To build the entire project, you will need the express.c source file from December, the exinterp.c and textsrch.c files from January, and the several source files introduced this month.

TEXTSRCH is a text retrieval system that provides a concordance-like index into a text data base. You provide the files of text, and TEXTSRCH builds an inverted word index into the files. Later, when you go looking for some forgotten reference to something, you use the TEXTSRCH query language to compose a query, and TEXTSRCH finds and reports to you the files that match your search. I described the syntax of the query two months ago and developed its interpreter last month. This month we build index files and connect it all together.

TEXTSRCH consists of several widely used techniques. Besides its usefulness as a text management tool, it provides examples of hashing, binary trees, command-line parsing, expression parsing, infix and postfix expression notation, and expression interpreting.

Listing One, page 140, is textsrch.h, Version 3. We built the first version two months ago and added to it last month. Most of this month's additions are function prototypes for the code that follows, but there is one other notable addition. The MAXFILES variable will influence the size of your data base index. It specifies the maximum number of text files that a TEXTSRCH index can support. If you change this number, you will change the size of the bit map that the expression interpreter needs. Remember, the bit map has one bit per file in the data base. When the bit map size changes, so does the size of one of the index files. The file named "TEXT.NDX" includes, among other things, a copy of the bit map that corresponds to every unique word in the data base.

Before we get into the index code, let's look at two general-purpose functions that TEXTSRCH uses and that you might find useful in other projects.

Parsing the Command Line

Listing Two, page 140, is cmdline.c, a source file that contains the *parse_cmdline* function. The function processes a program's command-line parameters and is the only part of TEXTSRCH that is oriented to any specific architecture in that it works with MS-DOS and Turbo C. You can easily redo it for another compiler or a different architecture. Its purpose is to parse file name specifications from the command-line into discrete file names and to call a specified function to process each file. In addition, *parse_cmdline* will sense the presence of command-line option switches and set entries in an associated array to logical true and false states accordingly. TEXTSRCH does not use this last feature, but *parse_cmdline* supports it nonetheless.

When your program first begins, it calls *parse_cmdline* and passes the conventional *argc* and *argv* variables that are available as parameters to the *main* function. Your program also passes the address of a 128-byte array of switches and the address of a function. The function, provided in your program, will process the files, one by one.

The option array contains the initial default setting for the command-line switches. This statement tests the array:

```
if (array('a'))
    . . .
```

If the command-line had *+a* on it, this test would be true. If the parameter was *-a* or if the default had a zero value in the 61st (ASCII assumed) position, the test would be false.

The function selects each file specification and calls the function addressed by the last parameter in the call to *parse_cmdline*. If you explicitly name files on the command-line, each of these is processed in turn. You can use wild cards or prefix a file name with the @ character to specify that the file names to be processed are recorded in a text file.

If you use wild cards, the files that match the ambiguous specification are processed in the order in which the directory scan finds them. I used the Turbo C *findfirst* and *findnext* functions for the directory scan. Microsoft C has similar but different ones named *_dos_findfirst* and *_dos_findnext*. Other compilers have their own versions of these functions or, at least, ways to call MS-DOS directly to use the MS-DOS functions that perform these operations. If you use a different operating system, then you must provide other ways to get the same result. If you use an operating system that does not support a directory scan or ambiguous file name specifications, then you must use the @ prefixed file of file names supported by *parse_cmdline*.

A Binary Tree

Listing Three, page 141, is bintree.c. It contains the functions to build and

search a binary tree. The tree in question is used for what we call "noise" words, and I'll explain that feature soon, but first you need to consider the *addtree* and *srchtree* functions and binary trees in general.

A binary tree is a search data structure that facilitates fast searches of key values. Each entry in the tree contains the value to be searched and two pointers — a pointer to the entries with values that collate higher than the current one, and a pointer to the entries that are lower. Even with a large number of entries, the number of compares needed to find a value is reasonably small.

The *addtree* function adds entries to the tree, which is empty at first. It allocates memory for the entry from the heap and then calls *srchtree* to see if the entry is already there or, if not, where it should be inserted.

After the tree is built, we use the *srchtree* function to tell us if a value is in the tree. In this implementation of the binary tree there are no other data values in an entry. We use the tree to determine the presence or absence of a value, nothing more.

Binary trees are most efficient when the entries are added to the tree in random sequence. If you were to build one from an ordered list of entries, the tree would have one long branch, and searches would be inefficient. There are tree balancing algorithms for binary trees, but we do not need one for our use here because we build the tree once, and our original order is random.

Noise Words

We use the binary tree for a list of noise words. Noise words are those words that are so common that we can assume that they appear in every file. The word "the" is a typical noise word. Because we can make that assumption, we can bypass such words when we build our index, thus saving index space. Likewise, we can bypass searching the index for such words during retrievals, thus saving time.

The *build_noisewords* function in bintree.c builds the binary tree from a text file that contains all the noise words. Listing Four, page 142, is noise.lst, a small file of noise words that I arbitrarily selected. You might want to review the list and modify it. The more words you can put into the noise word list, the more efficient your index will be.

Building the Index

Last month we stubbed the text search process by using the GREP utility program to process our queries. This month we toss that method aside and build an inverted index into our data base. To build the index we extract each word from each file in the data base. From the word we use a hashing algorithm to compute a random address. The random address points into SLOTS .NDX, a file of slots with one slot available for each possible random address. For this project we will assume a maximum of 64K words, so there will be 64K slots.

Each slot contains a long integer with the value *-1* if the slot is not in use. If the slot is in use, it contains a character offset into a text index file, named TEXT.NDX. The offset points to the first character of a variable-length record in TEXT.NDX. The record contains the matching text word, a file bit map, and a record chain pointer.

The bit map in the TEXT.NDX record contains one bit for each text file in the data base. If a bit in the bit map is one, the word appears in the file that is associated with the bit.

The file named "FILELIST.NDX" contains the names of the text files in the data base. The bit offset in the bit map and the name offset in FILELIST.NDX are the same.

The chain pointer in a TEXT.NDX record manages those words that hash to common slots. When a word hashes to a slot that a previous word used, we have a collision. We will add a new record to the end of TEXT.NDX and write its character offset into the pointer in the record pointed to by the slot entry. This builds a chain. The slot points to the first word that hashed to the slot. The first word's record points to the second, and so on. Subsequent collisions lengthen the chain.

The code that builds and searches these files is contained in Listing Five, page 142, index.c.

Building the index is managed by the program that starts with Listing Six, page 144, bldindex.c. It calls the *parse_cmdline* function to extract file names from the specifications on the command line and to execute the *index_file* function for each file to be indexed. The *index_file* function opens the specified file and calls the *extract_word* function (explained next) until there are no more words in the file to be extracted. The *srchtree* function tells us if the word is a noise word recorded in the binary tree. If not, we add the word to the index by calling the *addindex* function, found in index.c.

Extracting Words

To build an index (and to build the noise word binary tree) we need a function that extracts each word from a text file and normalizes it into a form suitable for indexing. The *extract_word* function in text.c, Listing Seven, page 144, does just that. You pass it the FILE pointer of an open file and a character pointer where it will copy the next word, and it does the rest. The normalization process consists of skipping characters that are not considered to be text and then selecting all text characters up to the next nontext character. For my purposes, I select character groups that include alphabetic characters, digits, the underscore (_), the plus (+) sign, and the pound (#) sign. This allows me to index text words, references to C++, function names, and preprocessor directives. The *extract_word* function converts alphabetic characters to lowercase. Other applications will have different requirements, so the *is-TEXT* macro in text.c allows you to define your own select criteria.

The Hashing Algorithm

Hashing is when you compute a random number from a key argument. You can then use the number as a random address into a file where the information related to the argument exists. With hashing you start with the key, compute the address, and use the address to find the record, all in the blink of an eye. Or so it would seem. In practice there is a bit more to it.

We will hash a random number from each of the words in our text files and later from the words we use as arguments in our queries. The algorithm I chose is a loose adaptation of the one that appeared in Steve Heller's "Extensible Hashing" article in the November 1989 issue of *DDJ*. It adds the seven-bit ASCII value of each character in the word to the initially zero hash total and shifts the total seven bits to the left. This add-shift loop continues for each character in the word.

There are as many different hashing algorithms as there are data formats to be hashed. The idea is to compute a random number that is evenly distributed across a defined range of values. If you can have 100,000 inventory records in a data base, the algorithm must compute from the inventory control key a random number between 1 and 100,000. A common technique is to reduce the key to a numerical value and divide that value by the prime number closest to the range. The remainder of that division is used as the random number.

In our approach we have decided that the range goes to 64K. Therefore, the 16-bit integer that we compute from a text string should suffice as a random number. Further reduction should not be necessary. Our 64K limit approxi-

(continued from page 124)
mates the number of different words we can expect to find in a text data base. Most works of prose will have far fewer than that number of unique words. Even if we build an index for some tome that exceeds that number, then our collision strategy ensures that every word has a place in the index.

The *compute_hash* function in index.c is our hashing algorithm.

TEXTSRCH Retrievals

We established our query language and interpreter in earlier installments. Now that we can build real indexes, we need to replace the stubs from last month with a real search process. Listing Eight, page 144, is search.c and it replaces last month's search.c stub program. The real thing is simpler than the stub because most of the work is now done in index.c. The search function calls *srchtree* to see if the word is in the noise word list. If so, the search function returns a bit map with all bits set to one because noise words are assumed to exist in all text files in the data base. If the word is not a noise word, the *search* function calls *search_index* in index.c to derive the bit map that says which files in the data base contain the word.

The *search_index* function calls *compute_hash* to develop the random address of the word's entry in SLOT.NDX. That entry contains the character offset to the first word that uses that slot. We read the offset and seek to that record location in TEXT.NDX. If the word in that record is the same as the one we are looking for, we have a hit. If not, and there are no further records chained to the slot, we have a miss. We navigate the chain if it exists and compare each word in it to the one we are searching for. If we get a hit, the bit map associated with the matching record is the one returned. Otherwise an all-zero bit map is returned.

The search process just described is for one word only. The expression interpreter from last month combines the searches for all the words in the expression with the Boolean operators and develops one bit map that represents the result of the full search.

The search.c file also contains the *process_result* function, which is no smarter than the one we stubbed last month. All it does is display on the console the name of the files that match the search criteria. What you will do with that list is up to you. See the discussion on Possible Enhancements for some thoughts on the matter.

Running TEXTSRCH

To build an index, get all your text files together and figure out how you are going to specify them on the command-line. If they are scattered throughout your subdirectories, you might want to make a list of them in a file. Use a text editor to make the file and put each file name on a separate line. These are the different commands for building an index:

```
bldindex *.txt
bldindex textfile.001 textfile.002
                                textfile.003
bldindex @files.lst
```

You can mix these formats like this:

```
bldindex *.txt textfile.001 @files.lst
```

The file specifications can have paths.

If the files named SLOTS.NDX, TEXT.NDX, and FILELIST.NDX already exist, you will be adding to an existing index. Try not to add files that you already indexed because this practice results in unnecessary redundancy in your index files. If the files do not exist, you are building a new index.

Building an index takes a long time, so you might want to do it in small increments, a few files at a time. A

power failure could make you start all over. The text files need to be accessible during the build phase, but can be elsewhere during retrievals. The retrieval process only identifies the file specifications as they existed when the index was built. Retrievals do not involve the text files themselves.

To perform a retrieval run the *textsrch* program from where the three .NDX files can be read. Enter query expressions as you did last month, and review the file lists that result. A null query expression terminates the program.

Possible Enhancements

TEXTSRCH has a lot of power and it has a lot of potential. As published here, it is a subset of a more powerful text processing system that I developed for engineering documentation applications. I use the version you see here with no enhancements for personal use, but there are several extensions that you might consider. Here are some of them.

Phrase Searches — TEXTSRCH retrievals are based on word searches because the index is built from discrete words. You can approximate a phrase search by combining all the words in a phrase into a query expression with the AND Boolean operator. This retrieval would deliver the names of the files that contain all the words in the phrase, but it would not tell you if the phrase itself was in any, all, or none of the files. You could search each file for the specific phrase itself by using Boyer-Moore or a similar text-matching algorithm. This approach would require modification of the expression input function to recognize the difference between phrases and individual words. The *process_result* function must be extended to do the follow-up search of the text files themselves.

Wild Card Searches — TEXTSRCH searches on a simple case-insensitive word pattern. You could add the "?" wild card by changing the search function in search.c to perform successive searches using every possible character where the wild card appeared.

Integrate with a Word Processor

The industrial system from which TEXTSRCH derives is integrated with a word processor. When the file list is presented to the user, he or she selects one, and the program fires up the word processor telling it to load the selected file and position itself to the first occurrence of the matching word or phrase. This implementation binds the two applications together and is beyond the scope of this column.

What Good Is It?

How might you use TEXTSRCH? I spend a lot of time reading and writing electronic mail and on-line service forum conversations. Sometimes I need to go back and find a message that mentioned something I want to recall. You know how it goes. Someone makes a comment about something that barely catches your attention. A few months later you are working on a new project and that comment would be helpful if only you could remember who made it and what they said. With TEXTSRCH indexes into my old mail I can usually find the messages that pertain to the subject in question.

I keep all my columns, articles, and book manuscripts stored away in TEXTSRCH data bases. That's right, I can't always remember what I wrote, when I wrote it, or where, and TEXTSRCH helps my failing memory (always the second thing to go).

I do another indexing trick with TEXTSRCH. I build dummy manuscripts that simulate the dozens of articles, manuals, and books I read every month. The dummy manuscripts contain nothing more than key words from the articles and books. The subsequent TEXTSRCH indexes help me find the hard copies where the material appears. Someday when books and articles are available in machine-readable media, or when I have a reliable OCR scanner, I'll be able to build the indexes directly and bypass the dummy manuscript process.

There is one other use that will be of interest to C programmers. If you put only the C keywords (*int*, *long*, *typedef*, and so on) into the noise word list and build an index from the source code for a huge project, you have in TEXTSRCH the beginnings of a handy cross reference of the functions and variables. It isn't worth the trouble if you are developing a new system, but if you have inherited one of those behemoth undocumented applications to maintain, TEXTSRCH might save you a lot of GREP time.

Book Review

Programming in C++ by Stephen C. Dewhurst and Kathy T. Stark. This book is a must for anyone who embarks on an intensive study of C++. It would help if you already had an exposure to C++, and knowledge of C is a prerequisite. I think I would have found this book a bit overwhelming a few months ago before I began to study C++. When read from the perspective of prior knowledge, however, this book is a gem. My one criticism is that it shares a characteristic found in most C++ and object-oriented literature. It uses fruit for examples. If I have to stomp through one more orchard of apple and orange objects, I fear I may be felled by the dreaded canker. To their credit, D&S don't dwell all that much on the ubiquitous citrus hierarchies, and other than for that one small lapse, their book is very good. To my surprise I found that the book has clear and concise explanations of procedural programming, top-down design, and structured programming, not OOP subjects at all. Most OOP books simply assume that you already know about these things. Well, you should, but if you need brief explanations of them written in a style that real people can understand, this book does the job.

And, bless them, D&S do not dwell on object-oriented programming as the be-all, end-all that we keep hearing about. No, they defer any detailed mention of OOP until Chapter 6, and then spend just a short amount of space with it. Skip that chapter and you might conclude from this book that C++ is simply an improved, extensible C. And that conclusion would not be far off the mark. Other parts of the book discuss those properties of C++ that combine to make it embraceable by the OOP ilk, but these presentations are not OOP-thumping paradigm evangelizations but rather mere explanations of the features of the paradigm as they are implemented in C++.

C and C++ will converge. Despite the minor differences that cause incompatibility between ANSI C and C++, a common language will evolve if only through usage. C++ brings too many improvements to C for that not to happen.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 140.)

Vote for your favorite feature/article.
Circle Reader Service **No. 10.**

(continued from page 117)
most beneficial option is to rearrange the sequence of instructions so that a useful instruction, which does not depend on the load data, is placed in the load-delay slot. In this case the load instruction takes only one clock and causes no disruption to the pipeline. The second option, if no suitable instruction for the load-delay slot can be found, is simply to order the instructions sequentially. When the register operation attempts to read the data from the register being loaded, the processor will freeze for one clock and then proceed. The i860 keeps track of which register has a load pending by way of a scoreboard. Although most loads will be cache hits, the scoreboard technique has further utility in the case of a cache miss. Instructions can proceed following a cache miss load until an instruction specifies the pending register. Programs can benefit by placing the load instructions as far away as possible from instructions that operate on the data.

The store instructions write data from a register to memory. For the i860, this can result in a write to the cache or a write to main memory. In both cases, the processor's pipeline does not have to wait for the write to complete. For cache hits, the new data is updated in the cache. For cache misses, the data is written to the on-chip write buffers, and the bus control unit carries out the memory write.

Addressing Modes

The integer load and store instructions access memory with one addressing mode that emulates several common ones. The basic load/store instruction format is shown in Figure 4 where *src2_reg* and *dest_reg* can be any of the 32-integer registers. The *src2_reg* is the base address, and *src1*, the offset, is added to it. For store instructions, *const* is a 16-bit offset constant that is embedded in the instruction. Load instructions also allow *src1* to be another one of the registers.

The instruction can specify data of 8-, 16-, and 32-bit values. For 8- and 16-bit values the operation occurs with the lower bits of the register. The *.x* designator in the instruction is set to *.b*, *.s*, and *.l*, according to the data size. Data must be aligned in memory to correspond to the effective address boundary (that is, 32-bit values on 32-bit address boundaries).

The integer register *r0* always contains the value 0. This aids in implementing multiple addressing forms without different instructions. The load instructions in Figure 5 show direct mode, register indirect mode, based mode, and based index mode addressing.

Table 1 is a complete list of the i860 core instructions. The i860's RISC core is also responsible for performing the memory operations for the floating-point registers. This allows the RISC core to keep the floating-point execution units fed with data, as the processor's architecture allows both a core and a floating-point instruction to be executed each clock. Floating-point memory access has an additional addressing mode.

Integer Operations

Once data has been loaded into the integer registers, any of the integer operations can be performed. These register operations are performed in one clock, and the result can be used as a source in the instruction that immediately follows. The i860 includes arithmetic, shift, and logical instructions and uses the form *operation src1, src2_reg, dest_reg*. The three operand-style instructions allow the operation to specify two source registers (or a source register and an immediate for *src1*) and to store the result to a third register without destroying any of the source values. This saves the program from copying a source value to a temporary register before the operation.

The add and subtract instructions allow an immediate value to be used as the subtractend or the minuend. For example, *r6 = 2 - r5* is encoded as *subs 2,r5,r6* and *r6 = r5 - 2* is encoded as

```
ld.x src1(src2_reg), dest_reg          ; dest_reg <- memory[src1 + src2_reg]
st.x   dest_reg,  const(src2_reg)      ; memory [const+src2_reg] <- rdest
```

Figure 4: *Load/store instruction format*

```
ld.l 8(r0), r15          ; r15 <- memory[8]
ld.l 0(r14), r15         ; r15 <- memory[r14]
ld.l 8(r14), r15         ; r15 <- memory[8 + r14]
ld.l r13(r14), r15       ; r15 <- memory[r13 + r14]
```

Figure 5: *Direct mode, Register indirect mode, Based mode, and Based index mode addressing load examples*

Core Unit	
Mnemonic	**Description**
Load and Store Instructions	
ld.x	Load integer
st.x	Store integer
fld.y	F-P load
pfld.z	Pipelined F-P load
fst.y	F-P store
pst.d	Pixel store
Register to Register Moves	
ixfr	Transfer integer to F-P register
fxfr	Transfer F-P to integer register
Integer Arithmetic Instructions	
addu	Add unsigned
adds	Add signed
subu	Subtract unsigned
subs	Subtract signed
Shift Instructions	
shl	Shift left
shr	Shift right
shra	Shirft right arithmetic
shrd	Shift right double
Logical Instructions	
and	Logical AND
andh	Logical AND high
andnot	Logical AND NOT
andnoth	Logical AND NOT high
or	Logical OR
orh	Logical OR high
xor	Logical exclusive OR
xorh	Logical exclusive OR high
Control-Transfer Instructions	
trap	Software trap
intovr	Software trap on integer overflow
br	Branch direct
bri	Branch indirect
bc	Branch on CC
bc.t	Branch on CC taken
bnc	Branch on not CC
bnc.t	Branch on not CC taken
bte	Branch if equal
btne	Branch if not equal
bla	Branch on LCC and add
call	Subroutine call
calli	Indirect subroutine call
System Control Instructions	
flush	Cache flush
ld.c	Load from control register
st.c	Store to control register
lock	Begin interlocked sequence
unlock	End interlocked sequence

Table 1: *i860 core instructions*

(continued from page 128)
adds *-2,r5,r6*.

Both signed and unsigned versions of each instruction are available. Add and subtract are also used to implement the compare function by specifying *r0* as the destination. For example *subs r4, r5, r0* will set the condition code (CC) if the contents of *r5* are greater than those of *r4*. The CC is used for the conditional branch instructions that are discussed later.

The logical instructions include the *AND*, *ANDNOT*, *OR*, and *XOR* operations and can be used to implement bit operations. For bit operations an immediate is used as *src1* with a 1 in the bit position to be operated on and zeros in the other bit positions. In addition to performing the operation, the logical instructions set the CC if the result is zero.

Because an instruction has only 32 bits, 32-bit constants cannot be embedded in a single instruction. Moving a 32-bit value into a register uses the special high version of the logical instructions that is indicated by the *h*. For example, the 32-bit hex value 9A9A5B5BH is moved into *r5* by first loading the lower half of the register and then using the *orh* instruction to modify the upper half of the register.

```
or 0x5B5B, r0, r5 ; r5 <- 5B5BH
orh 0x9A9A, r5, r5 ; r5 <- 9A9A5B5BH
```

The final class of integer operations is made up of the shift instructions. The i860 can barrel shift up to 31 bit positions in one cycle. The number of bit positions to shift is specified in *src1*. The shift right instruction also loads the *src1* into a special field in a control register. This field is used by the double shift instruction which concatenates two registers and shifts them into a third register. A rotate operation is performed by designating the same register as both *src1* and *src2* for the double shift.

Although the assembler allows you to specify a move instruction, the i860 does not need a separate move opcode. A shift instruction is used to implement the register-to-register move as it does not affect the condition code. The assembler will allow you to spec-

Religious Artifacts and Code Museums

Hal Hardenbergh

In the newer generation of U.S.-made micros, there is not a single CISC chip. The latest x86 and 680x0 chips are in fact code museums, not CISC chips, intended to support their enormous software bases. (The excellent term "code museum" was apparently coined by Gordon Bell.) The reason they have, for instance, fewer registers than one would like in 1990 is because they have to provide binary compatibility for code written for the 8088 and 68000 back in 1980. The 32532 is a code museum for code developed for the 16032, and the Z80000 is a code museum for Z8000 code.

The chief competitors of the code museums, we are led to believe, are the artifacts of a new religion called RISC. The two processors (SPARC and 29000) that most closely follow the RISC religion thereby require 32 multiply-step instructions to perform an integer multiply, plus up to eight more clocks for a trap or function call. Remember, nothing else can happen in those 40 clocks. Some programmers working with the latest "high performance" SPARC unit, the 330, say that it is a Pig when doing integer arithmetic.

Other chips usually called RISC processors do not in fact follow the RISC philosophy with respect to integer multiplies. The MIPS chip has a special functional unit that performs the multiply independently (other instructions can proceed during this multiply) with a latency of eight clocks. The 88000 uses its floating-point unit to perform integer multiplies; again, instructions can proceed in parallel. By not following the RISC philosophy, MIPS and the 88000 gain a significant performance advantage.

The RISC followers would have you believe that they alone try to make instructions run in the fewest clocks. Bull puckey. Worse, RISC zealots brag about their super-efficient load-store architectures. Hah. The i486, a mere code museum, performs push/pop operations 2.5 times faster than Sun's latest and highest-performing SPARC system (the 330). Why? One reason is that the i486 uses part of its budget of 1.2 million transistors to perform the register increment/decrement in parallel. This violates the RISC philosophy, so the SPARC and other RISC chips don't do it.

In other words, some of the "RISC features" are not exclusive to RISC, and some of the features that are exclusive to RISC degrade performance.

Are the i486 and 68040, then, the fastest possible chips? No; they're the fastest code museums that Intel and Motorola could make. Because they have a huge number of active devices, they are almost as fast as the conventional 32-bit RISC chips, which have the significant advantage of larger register sets and an instruction set optimized for the 32-bit world rather than a 16- or 8-bit world.

The simple fact is, if you use that 1.2 million transistor budget to build a device that does not have to support ancient code, you can build a hell of a fast device. Much faster than the SPARC, MIPS, 88000, or 29000, all of which have comparatively modest silicon budgets. Intel has proved this point with the i860. The i860 is not a religious artifact or code museum, but a very fast processor, looking sometimes like a RISC chip, sometimes like a CISC chip, and sometimes like a DSP chip.

The Intel i860 routinely performs single-precision floating-point math from four to twelve times faster than

Hardware engineer Hal Hardenbergh follows developments in microprocessor technology closely. He also (but not often enough) writes about his conclusions, which are always entertaining, frequently outrageous, and usually right on target. The following essay on RISC, CISC, and the Intel i860 came to us shortly after Hal had a chance to evaluate the i860.

ify *mov r3, r4* and implement it as *shl r0, r3, r4.*

Branch and Call Instructions

Instructions that change the sequence of program execution have long been the nemesis of pipelined machines. For many processors, these branch instructions require that the pipeline be flushed and restarted from the new branch target address. Because branches happen frequently, RISC processors use a delayed branch instruction where the instruction following the branch is executed before the branch takes effect. This allows the processor to continue the execution of a useful instruction while it begins fetching the new instruction from the branch target. The branch delay slot can be filled with a useful instruction from the block of code leading up to the branch; otherwise the target of the branch instruction can be moved to the delay slot and the target adjusted. The operation of delayed branches causes the execution order of code to differ from the assembly language sequence, as shown in Figure 6.

The i860 includes four unconditional delayed branches: *br, bri, call,* and *calli.* The *br* (branch) and *call* instructions allow a 26-bit offset as part of the instruction. The offset is in units of instructions, not in bytes, allowing a 256-Mbyte range. The *bri* (branch indirect) and *calli* (call indirect) instructions use the contents of a register as the target, thus allowing a full 32-bit address specification. In addition to changing the instruction flow, the call instructions save in *r1* the address of the second instruction after the call (the one directly after the call is the delay instruction). This is used as the return address for the subroutine by specifying *bri r1.*

The conditional branches that rely on the CC are the *bc* (branch on condition) and the *bnc* (branch on not). These include both delayed and non-delayed versions; the delayed version is indicated with a *.t.* At compile time, the programmer can usually predict if a conditional branch is going to be taken or not taken more frequently. If a conditional branch instruction is more likely to be taken, such as at the bottom of a

the MIPS or the 88000, even though both of those chips are capable of initiating a single-precision floating-point operation on every clock cycle.

How can this be?

How to Build a Fast Chip

The answer to that question is the real issue facing the next generation of microprocessors. The answer is concurrency. To run fast, you want a hell of a lot of concurrency. You get a hell of a lot of concurrency by having a huge silicon budget. The i860 has a huge silicon budget. The i860 has a hell of a lot of concurrency.

Why are the MIPS and 88000 chips limited to a throughput of about seven single-precision MFLOPS at best, even though they are in principle capable of issuing instructions at a 25-MFLOP rate? Simple. On those clocks where they issue a load or store, they cannot issue a math operation. And because, in conformance with RISC theory, they don't have an autoincrement address mode, they also don't issue a math operation when incrementing the address pointer or index. (There are other issues: Slower clock, 32-bit external data bus, no on-board data cache. But the biggie is the lack of concurrency.)

The i860, which is sampling now at 33 MHz and will run at 40 MHz in its production version, can perform all of the following in a single clock:

1. Execute a 32-bit floating-point multiply
2. Execute a 32-bit floating-point add or subtract

3a. Initiate a 64-bit floating-point register load or store that will take two clocks, or

3b. Initiate a 128-bit floating-point load or store, taking four clocks to finish

4. Increment, by an arbitrary amount, the address pointer that was just used for the load or store operation in (3a) or (3b)

As a result of the above, the i860 can do 21-bit convolutions (a common image-processing task) at 78 MFLOPS throughput. Not six or seven. Seventy-eight. Performing back propagations, FFTs, or matrix inversions, it will run at about 36 MFLOPS.

The fastest SPARC workstation that Sun is now shipping, the model 330, takes about 40 clocks to perform an integer multiply. The model 330 runs at 25 MHz, so that's 1.6 microseconds. During those 40 clocks, nothing else goes on in the integer execution unit. In that same 1.6 microseconds, the i860 can:

- perform 64 32-bit multiplies
- perform 64 32-bit floating-point adds
- do 256 bytes of memory I/O
- and perform all autoincrement addressing to support that memory I/O

And get this: If you use 128-bit load/stores for your I/O, the integer unit has 48 clocks left over to do something else while all this is going on. All this while the SPARC is executing one integer instruction. Isn't RISC wonderful?

True, the i860 is only good for floating-point number-crunching and (mostly) 3-D graphics acceleration. But Intel is coming out with a new chip, the superscalar i960. This is a completely new design, with a 64-bit external data bus, some elaborate schedulers, and three on-board integer-only execution units. This gives the new i960 a peak performance of 66 MIPS at a 33 MHz clock rate. No, those two numbers are not reversed; the i960 will perform two instructions per clock. The third execution unit is provided so that the unit can catch up if the scheduler postpones an instruction because of a resource (register) conflict.

You'll be able to purchase a workstation that uses the i960 for the main CPU and an i860 for floating-point and graphics for under $15K.

Intel paid for the performance of the i860 in cash. With its R&D budget of 86.6 million dollars per quarter, Intel is spending $780 million dollars per generation of microprocessors out of its own pocket. Intel has also received substantial funds ($100 million or so) from Siemens to develop the original i960 chip. That's why Intel was able to introduce two microprocessor designs, each with about 1.2 million active transistors, in a two-month period. And the new superscalar i960 will be introduced later this year. Where did the money come from? Why, from profits generated by the 386 family of chips, of course.

Hal called after writing to us about the i860 to tell us about the processor recently announce by Sharp. What he had to say was preliminary, but sounded interesting: 800,000 transistors, 4K internal scratchpad RAM, and . . . a throughput of 400 single-precision MFLOPS?? That's just ten times the throughput Hal says you can expect from an i860. — Eds.

loop, the delayed form should be used. For cases in which the branch is likely not to be taken, the non-delayed version allows more efficient coding. During execution, when the delay version is taken or the non-delay version is not taken, no disruptions are caused in the pipeline. A one clock penalty is incurred when the code guesses incorrectly.

By choosing an integer or a logical operation followed by a *bc* or a *bnc* instruction, all of the needed branch idioms can be implemented. There is also a non-delay branch instruction that branches on a compare-for-equality operation. The *bte* (branch if equal) and *btne* (branch if not equal) operations do register-to-register comparisons (or a register compare with a 5-bit immediate). Either of these branches can replace two instructions where appropriate, but at the expense of the offset being reduced to 16 bits.

Finally, there is a loop control instruction, called *bla*, that uses its own condition code called Loop-Condition-Code (LCC). The *bla* instruction is a delayed branch that performs a conditional branch-on-LCC, an add, and updates the LCC in the same instruction.

```
/* convert days & hours into hours */
/* C code */
int convert(days, hours)
   register unsigned int days, hours;
{
   unsigned int total;
   total = days * 24 + hours;
   return (total);
}
/* Compiler generated asm code */
          .file          "hours.c"
_convert:
          shl            2,r16,r28
          subs           r28,r16,r16
          shl            3,r16,r16
          bri            r1
           adds          r17,r16,r16
//_total  r16            local
//_days   r16            local
//_hours  r17            local
```

Example 1: *This conversion routine converts days and hours into total hours*

```
main()
{ int  sum,summer(),n,a[];
       *
       *
 sum= summer(a,8);
       *
       *
}
       int summer (a,n)
       int *a,n;
{      int i,sum=0;
       for (i = n-1; i >=0 ; i--)
           sum = sum + a[i];
       return(sum);
}
       .file    "sum.c"
                 *
       mov      r7,r16
       call     _summer
       or       8,r0,r17
       mov      r16,r17
                 *
                 *
_summer:
       mov      r0,r18
       adds     -1,r17,r17
       shl      2,r17,r28
       adds     r16,r28,r28
       adds     1,r17,r17
       adds     -1,r0,r20
       bla      r20,r17,.L65
       mov      r28,r16
.L65:
       bla      r20,r17,.L43
         nop
       br       .L42
            nop
.L43:
       ld.l      0(r16),r19
       adds      -4,r16,r16
       bla       r20,r17,.L43
       adds      r19,r18,r18
.L42:
       bri       r1
       mov       r18,r16
//_a   r16       local
//_n   r17       local
```

Example 2: *A subroutine called* sum_*ints that adds a series of integers*

Programming Examples

Now that we have looked at the basic instructions and instruction sequences, we can look at some simple yet revealing examples. Example 1 lists a conversion routine that converts days and hours into total hours.

The first optimization is that the parameters "days" and "hours" are passed in the registers *r16* and *r17* instead of being passed on the stack. This avoids having a pointer frame, or any entry or exit code. Second, the multiply by 24 is implemented as two shifts and a subtract. The first shift left by 2 implements a multiply by 4 and the subtract reduces it to a multiply by 3. The second shift left by 3 implements a multiply by 8 giving the total multiply of 24 (three times eight). Note that the first shift left takes advantage of the three operand instruction format, not destroying the original value in *r16*. This eliminates copying *r16* into a temporary register at the start of the routine, and allows the original contents of *r16* to be used as the source register in the subtract instruction that immediately follows. The final optimization is the add being performed in the branch-delay slot. The *bri r1* returns control to the calling routine with the result of the call returned in *r16*.

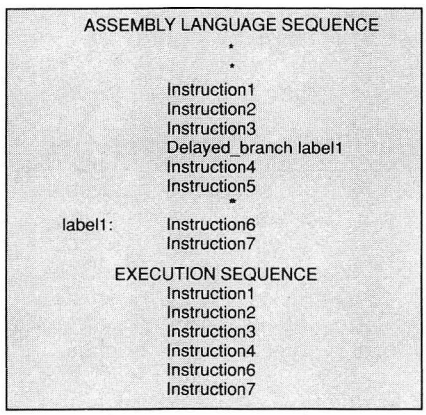


Figure 6: *The operation of delayed branches causes the execution order to differ from the assembly language sequence*

A subroutine called *sum_ints* that adds a series of integers is shown in Example 2. Because the integers to be summed are likely too numerous to fit in the registers, the routine is called with a pointer to the integers in *r16*. The other parameter, passed in *r17*, is the number of integers in the series. The example shows a loop where the data must be retrieved from memory.

The setup prior to the loop initializes LCC (with the *bla* instruction), checks that at least one loop iteration should occur, and moves zero into the sum register *r18*. Although the setup portion of the program may be slightly less than ideal, the routine's performance is clearly dominated by the loop portion. The loop loads the data from memory, decrements the pointer to the next integer, performs the loop control, and accumulates the sum of the integers. These four instructions are arranged to avoid any freeze conditions. The data from the load is not operated on until the branch-delay slot. The use of *bla* replaces the two or three separate instructions that would be required for this loop.

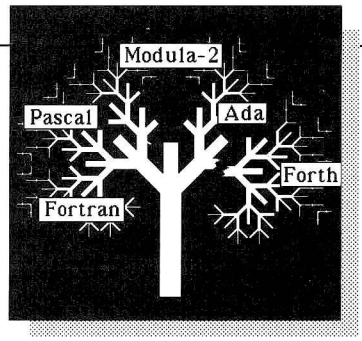
Although the compiler has done a good job of arranging the inner loop as a four-clock loop, it is not ideal. It is possible to use a loop index directly as the memory pointer and reduce it to a three-clock loop. An even more aggressive approach would be to unroll the loop to perform more loads and adds for each pass through the loop. This amortizes the loop overhead over a greater number of useful instructions. For the i860, a feature not discussed in this article, dual-instruction mode, allows the load and loop control to be overlapped with the summation performed in the floating-point registers.

Summary

In this article we have seen how RISC instruction sets are designed for fast, pipelined execution. We have seen how simple RISC instructions operate and how these instructions can be sequenced to perform various functions and reduce freeze conditions. Although most programs will be written in high-level languages, there is always the opportunity to check the compiler's output for efficiency, or to code the most time-critical routines by hand.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 8.**



The Day the Earth Wouldn't Sit Still

Jeff Duntemann, KI6RA

First, you freeze.

The little jitters happen out here now and then. They only last a second, and most of us just stop where we are and wait. At 5:04 I was outside on the new redwood front steps with a tube of Plastic Wood in my hands, filling in a syntax error I had made with the router.

But this jitter didn't pass. Out of the earth came a noise unlike anything in creation, a thing so deep and so pervasive that it bypassed the ears and went straight to the guts. What was a jitter became a violent, unending spasm, the earth boiling amidst the sound of two continents dragged one over the other. I dropped the plastic wood and held onto the stairs. From inside the house, I heard things begin to break: First glass, then the crockery of our water dispenser, then something large and heavy collapsing in the garage. Finally, with the horror in full sway, the agony of my house joined the agony of the earth, and it went on, and on, and on . . .

Twenty seconds passed, and the world grew quiet again.

The Machine Stops

Very quiet. Scotts Valley was dead. The silence was absolute, made sharper by the contrast with the all-consuming cacophony that had come before it. No birds, no cars on Highway 17, no tumbling burr from the sawmill, no susurrus from Seagate's many nearby buildings.

My three dogs stood like statues on the asphalt; I dove for them just as panic broke in them, and caught Mr. Byte and Chewy in time to throw them into the back of the Magic Van. Max, on the other hand, was already far down the driveway at full speed and was gone for many hours.

My water heater had broken its pipes, and the laundry room was filling with water.

In the east, a plume of black smoke was roiling into the sky. Somewhere there were sirens. I heard a woman yell in the distance. Another answered her. By 5:06 the silence had ended, and the long night had begun.

Being four miles from the epicenter of a Richter 7.1 temblor can teach you a few things. Technology is fragile. In an instant the power lines and phone lines were swept away. The town with more computers than people (and more disk drives than computers) simply stopped. No TV, no cable, no satellite downlinks. Ever the prepared radio amateur, I hooked a gel cell to an FM radio and tuned the dials. The local stations were not there. What did I think they ran on, wood stoves? A distant station crackled to me that unconfirmed reports of a violent earthquake in San Francisco were being investigated.

Right.

Over the days we learned the extent of the damage: Borland International's main building was deemed too dangerous to occupy. Much of Seagate's precision equipment was rendered useless. Hundreds of houses were utterly destroyed, including the home of Jack Davis, CEO of Metagraphics, maker of the MetaWindow graphics library. Thousands more were severely damaged. Downtown Santa Cruz was nearly destroyed.

A New Paradigm

Over the week we spent cleaning up, I did a lot of thinking. Our urban paradigm is very much one of the mainframe and the terminal. We are terminals connected to the power company mainframe, the telephone company mainframe, the natural gas line mainframe. It doesn't take much to make the mainframe go down, and when it goes down, the terminals go dead.

We need to stop being terminals and start being PCs. We need to plate every roof with amorphous silicon photovoltaic panels, and perfectly sealed battery technology to the point where a water heater-sized unit can operate the household for several days on a full charge and not cost as much as the house to buy. Homes should be power exporters during the day, charging their own cells to capacity and then feeding surplus energy onto a peer-to-peer power grid to run smaller installations without their own cells. If the grid goes down, the homes themselves remain livable refuges.

Similarly, every home should have a 12-inch precision parabolic dish antenna on the roof, aimed at a geosynchronous satellite with the bandwidth to handle an entire state at one gulp. Communication with the outside world should not hang from a fragile copper wire.

Home heating is a problem. Perhaps the answer is to break down water into hydrogen with solar current during the day and then burn the hydrogen for heat at night. We need to set the methane and propane and yes, my fellow Sixties radicals, even the "clean" wood stoves aside. Burning carbon — any carbon — feeds the greenhouse.

It makes you grin with the kind of grin that hurts: For the last hundred years we've been using technology to tie ourselves to abject dependence upon one another; now it's pretty plain that we'd better use that same technology to help each household stand alone. For the Earth he done spoke, as he does now and then. Even if it takes another hundred years, we need to change our paradigm to take the Earth into account.

Let's consider this one — the Pretty Big One — a warning.

Why Is There Modula?

It took the Mighty U.S. Post Office three days to resume the mails, but Federal Express was making deliveries the next day. We were still living out of the Magic Van when the FedX lady roared up the drive and dropped a box in my hands. In the box was a truly remarkable thing: Stony Brook Modula-2.

Now, at last count I had nine DOS-based Modula-2 compilers on my shelves here. All work; most work well; some work beautifully, and one — TopSpeed Modula-2 — works spectacularly. But perhaps latecomers have the edge, because there is a certain crackle of excellence that sets Stony Brook apart, even from the formidable TopSpeed. Stony Brook is worth a closer look, in the interest of understanding the Vice President of structured languages: Modula-2.

I've given Modula short shrift in this column, because President Pascal keeps hogging the spotlight. Modula-2 freaks

keep waiting for the Prez to kick off and let their man ascend to the top spot, but the Prez lives on. (Lordy, now there are even two of them.)

It won't happen. Turbo Pascal brought Pascal to the same sort of critical mass that Cobol and OS/360 achieved many years ago. It will literally never go away because there is so much of it out there. Fortunately, Pascal continues to evolve, and today it serves needs unimagined 18 years ago at its birth. Pascal may in fact absorb Modula-2 over time by stealing its best features one by one. The last chapter has definitely not been written.

(And Pascal's not the only one. I heard not long ago that a committee has convened to add object-oriented extensions to — hold your breath — Cobol. I can see it now: "File: Go read yourself. Record: Go write yourself." It is to boggle the mind.)

So why is there Modula? What good is it? And why hasn't it done better?

Good questions. Let's talk.

No Ambiguities

If I were to choose the #1 imperative of Modula-2, it would be this: Let there be no ambiguities. This imperative works in the small and in the large.

For example, in Pascal it's sometimes impossible to tell whether an element in an expression is a variable or a function. Like this: Is *MaskedMarauder* a variable or a function?

```
Zorro := MaskedMarauder; { Pascal
                           code }
```

The answer, of course, is that you can't tell. You have to go look further up the source code to see.

In Modula-2, a function identifier is always followed by a pair of parentheses, even when it takes no parameters. Like so:

```
Zorro := MaskedMarauder( );
                  (* Modula-2 code *)
```

Now the marauder loses his mask a little. C does this as well, which is one of C's few advantages over Pascal. (And a small one it is, too; rather like a diamond ring dropped into a fifty-year-old outhouse. You dig for it . . .)

In the larger view, Modula-2 always lets you know where every program element is defined. Apart from a small suite of standard functions and control structures, everything used by a Modula-2 program must be listed at the top of each module, in statements that describe where each element comes from:

```
FROM SuperHeroes IMPORT
                    MaskedMarauder;
FROM Japan IMPORT
                    Rice,Cars,VCRs;
FROM LawSchool IMPORT
                    SmallMinds;
```

Sometimes, in a largish module of a largish program, there may be dozens of such statements.

This is one very full step past the modularity mechanisms of Turbo/Quick-Pascal, which tells you which modules (in Pascal, units) are referenced but not what was taken from them:

```
PROGRAM UnholyMess;
USES SuperHeroes,Japan,LawSchool;
```

This doesn't tell me where the identifier *SmallMinds* comes from, though depending on your politics you might intuit either *Japan* or *LawSchool*. (I didn't use unit *Congress* here or it'd be a no-brainer.) In Modula you don't have to intuit, or look it up in the interface section of every module you use. The importer tells all. And in a complex program this can save enormous amounts of source code scanning!

Sometimes this quest for unambiguity makes things a little bit inconvenient. In Modula-2, the type and number of parameters in a procedure can never change. Out goes Pascal's *Read* and *Write*, which can take any number of

parameters in numerous modes. Instead, you have a separate output and input procedure for each data type, such as *WriteString*, *ReadInt*, *ReadCard* (for the *CARDINAL* unsigned integer type), and so on. What might be done in a single *Writeln* statement in Pascal becomes a conga line of *WriteThis*, *WriteThat*, and *WriteTOther*, one for each different type you need to output. Whether this is a bug or a feature depends on whether you write applications or compilers; Pascal's open-ended I/O routines make matters lots trickier for compiler architects.

So Modula-2 eliminates a great many of Pascal's ambiguities, but, oddly, it doesn't go nearly as far in one respect as it might. Numerous control structures end in the identical word *END*; what Wirth should have used is a unique terminator for each structure, such as *ENDFOR*, *ENDCASE*, *ENDWHILE*, and so on. Wirth wisely added an *END* word to the end of every *IF . . THEN* statement:

```
IF Richter < 8.0 THEN
      GrabSomething;
      HangOn;
   ELSE
      GrabSomething;
      SayPrayers
END;
```

(Note the welcome absence of redundant *BEGIN* words.) Still, what he should have added was a unique terminator word such as *ENDIF*. This would have added tremendously to the readability of heavily nested control structures. Ahh, well. Maybe Modula-3.

Flexibility Without Ambiguity

One of the critics' numerous gripes against Pascal is its rigid type checking. If you want to pass an array to a procedure, the formal parameter must be identical in type to the actual parameter. In other words, if you want to create a general-purpose routine for sorting arrays of records, in Pascal you must declare the formal parameter as having specific bounds. When called, the array passed to the sort procedure as its actual parameter must not only have elements of the same type as the formal parameter but identical bounds as well. A variable defined as

```
ARRAY[0 . . 1023] OF MyRecord
```

cannot be passed as an actual parameter to a sort procedure defined as

```
PROCEDURE SortEm(VAR Target :
                        RecArray);
```

where *RecArray* has this definition:

```
ARRAY[0..255] OF MyRecord
```

Rigid indeed. And not entirely true, even of Pascal. Standard Pascal contains a feature that is implemented so rarely (and not in either Turbo or Quick-Pascal, though present in MS Pascal as its "super array") that most people have just plumb forgot about it: The conformant array.

Modula-2 supports conformant arrays, though it uses the far more descriptive term "open array." A formal array parameter of a procedure is an open array when it is defined without bounds:

```
PROCEDURE SortEm(VAR Target :
                  ARRAY OF MyRecord);
```

Only one-dimensional arrays may be open arrays. Within the procedure, the high bound may be returned by calling the standard procedure *HIGH*. The low bound is always assumed to be 0. This way, you can pass an array of *MyRecord* of any size to *SortEm*, and simply ask the *HIGH* procedure for the index of the last element. Open arrays are an excellent example of how Modula-2 can simultaneously be more flexible than Pascal while remaining less ambiguous.

Modula-2 also provides standard port for coroutines, which try as I might defy description in anything smaller than a column unto themselves. We'll take them up again in the future. For now, suffice it to say that coroutines amount to poor-man's multitasking, and are of minimal usefulness — but are a giant step in the right direction.

And Some Boo-boos

The news on Modula-2 is almost entirely good. Just to make sure we were all awake, however, old Nick Wirth threw in some zingers. Worst of these is the rather ridiculous restriction on the number of elements in a set. A standard Modula-2 set may have only 16 elements. Wirth says he did this to make compiler implementation easier, which is an explanation but not an excuse. Somebody has to use these damned things, Dear Doctor! Probably 90 percent of all sets used in Pascal are sets of *Char*, in which there must be 255 slots. Out the window. (In standard Modula-2, at least. More later . . .) My hunch is that this restriction has sunk more conversions to Modula-2 than all its other peccadillos put together.

Compared to that, other irritations are minor. Modula-2 is case sensitive. I don't like it in C, and I don't like it in Modula . . . but programmers can and will live with it. Comments must be bracketed with the (* *) comment delimiters rather than the simpler curly brackets used almost universally in Pascal. Open arrays are limited to one dimension. There is no double-precision real number type, like *Double* or *Extended* in Turbo Pascal. In general, with regard to numeric calculations, Modula-2 is type-poor compared to Turbo Pascal.

But by and large, Modula-2 is a considerable improvement over Pascal. This is especially true when large projects must be broken up into modules. Modules, after all, were what drove the design of Modula and gave it its name. So why hasn't Modula done better than it has? There are two fundamental reasons:

- Turbo Pascal got there first; and
- Wirth forgot to define standard libraries

Overdue at the Standard Library

The first of these problems was simple fate, and a reflection of the power of momentum and source code critical mass. Had Philippe Kahn brought a super fast Modula-2 to America in his gym bag in 1982, Pascal would probably be about as popular today as JOVIAL or SNOBOL. Turbo Pascal is now Standard Pascal, regardless of how much those ISO people grind their teeth. When Microsoft anointed the Turbo Pascal Standard with QuickPascal last May, the game was over in the Pascal standards business.

The second problem was totally avoidable. When Wirth released the Modula spec back in 1980, he deliberately made it lean and mean — about as lean as a language could be without lapsing into anorexia. Pure Modula-2 is all bones and no flesh. Wirth defined the standard data types, modularity mechanisms, control structures, a handful of standard procedures and functions . . . and stopped.

Why? Portability, of all things, which in today's world of philosophically incompatible operating platforms isn't even as valuable as your average city planner and drops to the usefulness of a tobacco lobbyist. It's silly to argue with portability freaks and I won't try. But my position is plain: Drop-in portability is impossible. Least-common-denominator portability costs more than it is worth. The smart thing to do is choose a platform and make the most of it. When you must jump platforms, expect to rebuild everything you own from scratch, or you're strapping on a wooden leg and tying one hand behind your back.

One reason that Turbo Pascal caught on so quickly is that it made the most of its operating environment, DOS. It had built-in support for parsing and returning DOS command-line parameters, built-in dynamic string support, built-in 8086 memory addressing and I/O support, and plenty more. Version 4.0 added versatile (if not lightning-quick) graphics and the fabulous DOS unit. The value in such libraries is almost immeasurable: How many of you could duplicate the BGI on your own? 90 percent of you with your hands up, lower your eyes and go to confession.

Programmer man-hours cost more every year. Libraries save man-hours. Lots of them. Once he was satisfied with the basic definition of Modula-2, Wirth should have spent six months or so thinking about the sorts of utility modules that programmers spend the most time building, and come up with specs for a suite of libraries to serve those needs. With a little thought, most utility libraries can be divorced almost completely from the details of their operating platforms. A world-coordinate graphics system, for example, can be made to operate on any raster-based graphics system. Any file system needs library routines to scan for ambiguous file names, and most basic I/O operations can be specified without bowing to the details of a particular platform. Obviously, the details of implementation are closely tied to the platform beneath them, but the spec can with cleverness be made almost entirely platform-independent.

What bothers me most, I think, is that Wirth didn't even try.

So what we've seen is each Modula-2 compiler vendor implementing libraries to meet the needs of DOS programmers, and every one does things just a little bit differently, both in terms of how things are done, and also in terms of what is done and what is left out. The kernel of Modula-2 is so compact and simple that the most visible portions of any given implementation are its libraries, and these are so different from vendor to vendor that each implementation begins to look like a separate language. Modula-2 has become a Tower of Babel, and, sadly, (with a little foresight) Niklaus Wirth could have prevented most of that.

Stony Brook Specifics

Maybe this glum view will scare you away from Modula-2. I hope it won't. If you choose a solid implementation and stick with it, for your purposes the Babel Factor ceases to be a problem. There are two extremely good implementations for DOS these days, and either will serve you well. I've discussed TopSpeed Modula-2 in earlier columns, and I'd now like to focus on Stony Brook.

Editors and writers tend to notice documentation. Mark me well: Stony Brook has by far the finest documentation of any Modula-2 product that has ever crossed my desk. It is well written and well organized, and while I've found plenty of small quibbles (mostly in explanations that do not say nearly enough) the total effect is wonderful.

Stony Brook goes its own way in terms of a programming environment. Instead of centering on a single text file as the focus of a project, Stony Brook's environment centers on a subdirectory that is devoted to a single project consisting of several or many files. "Home base" in the environment is a screen listing of all component files belonging to that project. Modula-2 tends to produce a lot of files, because each module has a separate implementation and definition file. Stony Brook's environment lines them all up vertically, with a highlight bar selecting one as the current focus. By pressing enter, the highlighted file may be edited. By pressing other command keys, additional environment tools may be brought to bear on the highlighted file.

For example, it's often useful to know what modules are used by a given module, and also what modules use that module. Two keystrokes will display any given module's import list (what it uses) or client list (who uses it.) When I had to divide my attention among a great many separate but related module files, I found that this system works quickly and extremely intuitively.

Within the Professional package, there are two complete compilers: One designed for fast compilation, and the other designed to squeeze the last bit of performance out of the code. The global optimizations are impressive, both in terms of .EXE size and speed. An example: The non-optimized .EXE file for *JTerm* is 32,966 bytes in size. Run it through the optimizing compiler and it shrinks to 16,277. Switching from one compiler to the other is a single command, so you can develop rapidly by using the fast compiler, then generate a user-testable .EXE file by invoking the optimizing compiler.

Exploring the Libraries

Stony Brook's excellent documentation made it lots of fun to explore the libraries. In less than an hour I rigged up a simple interrupt-driven telecomm program that included the ability to capture incoming characters to a text file on disk — all in about 100 lines of code. The program is shown as Listing One, page 146, and it illustrates several of Stony Brook's utility libraries.

The telecomm session runs inside a bordered window. The window occupies the entire screen in *JTERM*, but it doesn't have to. The capture file is named "CAPTURE.TXT," and is either created (if it does not already exist) or opened to the end of data when the program is run. Function key F7 begins sending incoming characters to the file, and function key F8 suspends data capture. This allows you to "grab" specific messages of interest on a timesharing system such as CompuServe with minimal command-entry.

CommPort is an example of a superb library not present in any other Pascal or Modula-2 implementation I know of. It implements a suite of interrupt-driven serial port support routines that would be an unholy hassle to implement independently. (I've done it and lost some hair in the process.) Basically, you init a port, turn on interrupts, and check a buffer for incoming characters. Assuming you allocate a large enough buffer and check the buffer regularly, you needn't worry about losing characters. The hidden interrupt machinery saves them in the buffer, where they remain ripe for the picking.

JTerm might have been even simpler had the *Screen* library supported simple TTY-style output to a window. TTY output is available to the full screen, but within a window I had to fake it by using *Screen*'s various cursor-manipulation procedures. No serious problem, as Listing One will attest. *Screen* has excellent support of colors and attributes for text windows. Its window borders are color bars rather than line-drawn characters, which imparts an altogether different sort of look to Stony Brook's text windows.

The range of Stony Brook's standard libraries is absolutely stellar, and I've used only the most basic of them in *JTerm*. Several modules are provided to support all the most useful numeric/string conversions. Another module provides exit procedure support roughly equivalent to Turbo Pascal's. Modules are provided to query time and date from the system clock; to return the command line and the environment variables; to create, remove, set, and search directories; to set and read the logged disk drive and query available disk space; to support the standard Microsoft mouse API; to control the speaker; to shell out to DOS or run DOS programs and commands; to support both long and short heaps; to support variable-length strings (ASCIIZ without a length byte, sadly); and very complete support of long integers and long cardinals, including transcendental operations.

A graphics library is provided, but it doesn't have the range or flexibility of the BGI. Still, for simple graphing and charting it's more than adequate.

Modular Multitasking

One of the most intriguing aspects of Modula-2 is its hooks to multitasking platforms. Both TopSpeed and Stony Brook have similar libraries to support preemptive multitasking of processes. Under DOS this is done out of whole cloth, with a custom scheduler and all the attendant baggage glued onto DOS like antlers on a Doberman; yes, the beast can handle it but it looks funny. Under OS/2 (which both products support) the native facilities of the platform are used. How well multitasking works under DOS I have yet to test, but I will test it and report on it in a future column.

If you need to develop for OS/2, Modula-2 becomes a very attractive language. Products are available immediately (Turbo Pascal's OS/2 schedule has not been announced) and the language itself presents a cleaner interface to multitasking than Pascal or even C. Stony Brook bindings to the OS/2 Presentation Manager SDK are well along and may be complete by the time you read this. Bindings to the Microsoft Windows SDK are already shipping, and while I haven't tried them yet I hope to take a shot at it soon. I don't expect to take back my firm opinion that OOP is the way to develop for both Windows and PM, but certainly Stony Brook can't help but be a step up from the Microsoft SDK under C.

More Modula-2 code next time, along with further experimentation with the Stony Brook compiler.

Modula-2 Books

I have three shelves full of books on Pascal (and I keep by no means all of them) but only seven books on Modula-2, including a couple of questionable

Products Mentioned

Stony Brook Modula-2
Stony Brook Software
187 East Wilbur Rd., Ste. 9
Thousand Oaks, CA 91360
805-496-5837
QuickMod V2.0 for DOS $95
QuickMod V2.0 for OS/2 $95
Professional Modula-2
(includes DOS and OS/2 optimizing compilers plus both QuickMod compilers) $295

TopSpeed Modula-2
Jensen & Partners International
1101 San Antonio Rd., Ste. 301
Mountain View, CA 94043
415-967-3200
DOS compiler $99.95
OS/2 compiler $195.00

Programming in Modula-2, Fourth Edition
Niklaus Wirth
Springer-Verlag, 1988
ISBN 0-387-50150-9
Hardcover, 182 pages, $29.95

Modula-2 for Pascal Programmers
Richard Gleaves
Springer-Verlag, 1984
ISBN 0-387-96051-1
Paper, 145 pages, $23.50

Modula-2 Programming
John W. L. Ogilvie
McGraw-Hill, 1985
ISBN 0-07-047770-1
Hardcover, 304 pages, $29.95

Advanced Programming Techniques in Modula-2
Terry A. Ward
Scott, Foresman & Company, 1987
ISBN 0-673-18615-6
Paper, 300 pages, $21.95
Listings diskette $14.95

ones. Wirth's own defining volume, *Programming in Modula-2*, is not terribly useful unless you have nothing else. Too dry, too short, too bereft of useful examples. On the other hand, Richard Gleaves' *Modula-2 for Pascal Programmers*, while even shorter, has a unique mission: To teach you Modula-2 by way of its similarities to and differences from Pascal. This shouldn't be the only book you use to pick up Modula, but it should certainly be an accessory.

A good, general first book for learning Modula-2 is *Modula-2 Programming*, by John W.L. Ogilvie. The style is clean, the organization rational, and most of the examples are printed in a dark monospace font that is very easy to read. This book and the Gleaves book will get you there. Once you're there, getting better often comes (in addition to coding like crazy) from reading good code by somebody who knows his stuff. One book to pick up is *Advanced Programming Techniques in Modula-2* by Terry A. Ward. This book is mostly code, but it's good code, including set, string, sort, and search libraries, some user interface tools, and a simple expert system. (Publisher details on all books at the end of the column.)

NOTE: I do not recommend the book *Modula-2: A Seafarer's Guide and Shipyard Manual* by Edward J. Joyce. The type will make your eyes fall out, the writing is indifferent to bad, and it just doesn't contain the quantity and quality of information that the Ogilvie book does.

There's a dearth of good books on Modula-2. Most of the titles I have are between two and five years old. If any publishers reading this have any current Modula-2 titles in print, I'd appreciate seeing them, and perhaps will let the readers of this column know about the better ones.

Halloween II: From Hell to Arizona

Eek! It's Halloween night again, and I just remembered that I finished writing my first *DDJ* column exactly one year ago today. That makes this my 13th column (double eek!), most appropriate for describing earthquakes and other inappropriate geological behavior. Don't bother looking for outhouses to knock over; the quake beat all of us to it.

Carol and I have quaked our last, however, and as soon as the palace here sells, we are packing up dogs, books, machine shop, radios, and 386 boxes and heading for Scottsdale, where Keith Weiskamp and I will be launching our new magazine *PC Techniques* this spring. *DDJ* editor-in-chief Jon Erickson has graciously allowed me to continue this column, and I expect I'll be here again, same time same station . . . if here would just sit still long enough for me to make the dash to Arizona!

Write to Jeff Duntemann on MCI Mail as JDuntemann, or on CompuServe to ID 76117,1426.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listing begins on page 146.)

Vote for your favorite feature/article.
Circle Reader Service **No. 11.**

Listing One *(Text begins on page 123.)*

```
/* ---------- textsrch.h ---------- */

#define OK    0
#define ERROR !OK

#define MXTOKS 25           /* maximum number of tokens */
#define MAXFILES 64         /* maximum number of files  */
#define MAXWORDLEN 25       /* maximum word length      */
#define MAPSIZE MAXFILES/16 /* number of ints/map       */

#define SLOTINDEX  "slots.ndx"
#define TEXTINDEX  "text.ndx"
#define FILELIST   "filelist.ndx"

/* ---- the search decision bitmap (one bit per file) ---- */
struct bitmap {
    int map[MAPSIZE];
};

/* ------- the postfix expression structure -------- */
struct postfix  {
    char pfix;      /* tokens in postfix notation */
    char *pfixop;   /* operand strings            */
};

/* --------- the postfix stack ---------- */
extern struct postfix pftokens[];
extern int xp_offset;

/* --------- expression token values ---------- */
#define TERM    0
#define OPERAND 'O'
#define AND     '&'
#define OR      '|'
#define OPEN    '('
#define CLOSE   ')'
#define NOT     '!'
#define QUOTE   '"'

/* --------------- textsrch prototypes --------------- */
struct postfix *lexical_scan(char *expr);
struct bitmap exinterp(void);
struct bitmap search(char *word);
void process_result(struct bitmap);

/* -------------- binary tree prototypes ------------- */
void addtree(char *s);
int srchtree(char *s);
void delete_tree(void);
void build_noisewords(void);

/* ------------- command line prototypes ------------ */
void parse_cmdline(int,char **,char *,void (*)(char *));

/* ---------------- text prototypes ------------------ */
void extract_word(FILE *fp, char *s);

/* --------------- index prototypes ------------------ */
void open_database(void);
void init_database(void);
void close_database(void);
void addindex(char *word, int fileno);
void indexing(char *filename);
int getbit(struct bitmap *map1, int bit);
struct bitmap search_index(char *word);
char *text_filename(int fileno);
```

End Listing One

Listing Two

```
/* ---------- cmdline.c ---------- */

#include <stdio.h>
#include <dir.h>
#include <string.h>
#include "textsrch.h"

/*
 * Parse a command line:
 *  filename1 filename2 ... filenamen
 *  @filelist
 *  wild cards
 *  -+x option list (-a +b -c -xyz +pdq)
 *  any mix of the above
 */

/* ---- parse a command line for options and file names ---- */
void parse_cmdline(int argc, char *argv[], char *options,
                void (*func)(char *fn))
{
    char path[65];
    FILE *fp;
    while (argc-- > 1)  {
        switch (**++argv)   {
            case '/':
            case '+':
                /* -------- add an option --------- */
                while (options && *++*argv)
                    options[**argv] = 1;
                break;
            case '-':
                /* -------- remove an option --------- */
                while (options && *++*argv)
                    options[**argv] = 0;
                break;
            case '@':
                /* ----- a file of file path/names ----- */
```

```
                if ((fp = fopen(*argv+1, "rt")) != NULL)
                    while ((fgets(path, 65, fp)) != NULL)   {
                        path[strlen(path)-1] = '\0';
                        (*func)(path);
                    }
                break;
            default:
                /* ---- a file spec on the command line ---- */
                if (strchr(*argv, '*') || strchr(*argv, '?')) {
                    /* ------ an ambiguous file spec ------- */
                    struct ffblk ff;
                    char *cp;
                    int rtn;

                    /* ---- copy the ambiguous file spec ---- */
                    strcpy(path, *argv);

                    /* ---- find the filename part ---- */
                    if ((cp = strrchr(path, '\\')) == NULL)
                        if ((cp = strrchr(path, ':')) == NULL)
                            cp = path-1;
                    cp++;

                    /* ---- search for matches ---- */
                    rtn = findfirst(*argv, &ff, 0);
                    while (rtn == 0)    {
                        strcpy(cp, ff.ff_name);
                        (*func)(path);
                        rtn = findnext(&ff);
                    }
                }
                else
                    /* ----- an unambiguous file spec ----- */
                    (*func)(*argv);
                break;
        }
    }
}
```

End Listing Two

Listing Three

```
/* ------------ bintree.c ---------- */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "textsrch.h"

struct bintree {
    struct bintree *lower;
    struct bintree *higher;
    char wd[1];
};

static struct bintree *first, *next;

/* ---------- add a string to a binary tree ----------- */
void addtree(char *s)
{
    struct bintree *tp;
    int intree;

    tp = malloc(sizeof (struct bintree) + strlen(s));
    if (tp == NULL)
        return;
    strcpy(tp->wd, s);
    tp->lower = tp->higher = NULL;
    if ((intree = srchtree(s)) != 0)    {
        if (first == NULL)
            first = tp;
        if (next != NULL)   {
            if (intree < 0)
                next->lower = tp;
            else
                next->higher = tp;
        }
    }
}

/* ------ Search a binary tree for a string.
    Return 0 if the string is in the tree.
    Return < 0 or > 0 if not.
    If not, next -> the node where insertion may occur. --- */

int srchtree(char *s)
{
    struct bintree *this;
    int an = -1;
    this = next = first;
    while (this != NULL)    {
        if ((an = strcmp(s, this->wd)) == 0)
            break;
        next = this;
        this = ((an < 0) ? this->lower : this->higher);
    }
    return an;
}

/* ------- delete the nodes of a branch of the tree ------ */
static void delete_nodes(struct bintree *nd)
{
    if (nd->lower)
        delete_nodes(nd->lower);
    if (nd->higher)
        delete_nodes(nd->higher);
    free(nd);
}

/* ------- free all memory allocated for the tree -------- */
void delete_tree(void)
{
    delete_nodes(first);
}
```

(continued on page 142)

Listing Three *(Listing continued, text begins on page 123.)*

```
/* ------- build a binary tree of noise words -------- */
void build_noisewords(void)
{
    FILE *fp;
    char word[MAXWORDLEN+1];
    /* ---- open the noise word file ---- */
    if ((fp = fopen("NOISE.LST", "rt")) != NULL)    {
    /* ---- extract words and add them to the list ---- */
        while (!feof(fp))   {
            extract_word(fp, word);
            /* -------- search the noise word list -------- */
            addtree(word);
        }
        fclose(fp);
    }
}
```

End Listing Three

Listing Four

```
so very what he she her both there if above only it again
done our then from just my along left who may we all these
them us after once need through onto others can want where
would nor none with do here been was see own since off this
not will which itself that each to take down on you under at
their however the an as even have whose a said before or
nearly below are those possible alike begin out than having
two know when way often together by many thus whatever i his
past another though other entire in against am most taken
instead him because for might too rather few soon either near
about every until neither beyond your better must usually
several does such something using put more whether sent any
later like and now they also upon close be use of is should
could were into made further used let how enough its himself
known never become sure over next up among had causes has no
old some come but while me
```

End Listing Four

Listing Five

```
/* ----------- index.c ------------- */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "textsrch.h"

static void setbit(struct bitmap *map1, int bit);
static unsigned compute_hash(char *word);

static FILE *slots, *text, *flist;
static char path[65];
int file_count = 0;

/* ------ open or create the index files for building ------ */
void open_database(void)
{
    unsigned ctr = 0xffff;
    long empty = -1L;
    build_noisewords();
    if ((slots = fopen(SLOTINDEX, "r+b")) == NULL)
        slots = fopen(SLOTINDEX, "w+b");
    if (slots != NULL)  {
        if ((text = fopen(TEXTINDEX, "r+b")) == NULL)
            text = fopen(TEXTINDEX, "w+b");
        if (text != NULL)   {
            if ((flist = fopen(FILELIST, "rt")) != NULL)    {
                while (fread(path, sizeof path, 1, flist))
                    file_count++;
                fclose(flist);
                flist = fopen(FILELIST, "at");
                return;
            }
            /* ---- if the file list does not exist,
                we must be building a new data base ----- */
                if ((flist = fopen(FILELIST, "wt")) != NULL) {
                    /* --- preset the slots to -1 values --- */
                    printf("\nBuilding index. Please wait...\n");
                    while (ctr--)   {
                        if ((ctr % 1000) == 0)
                            putchar('.');
                        fwrite(&empty, sizeof(long), 1, slots);
                    }
                    file_count = 0;
                    return;
                }
                fclose(text);
            }
            fclose(slots);
        }
        printf("\nCannot establish index files");
        exit(1);
    }

    /* -------- open an index data base for retrieval ------- */
    void init_database(void)
    {
        build_noisewords();
        /* ---------- open all three index files ---------- */
        if ((slots = fopen(SLOTINDEX, "rb")) != NULL)   {
            if ((text = fopen(TEXTINDEX, "rb")) != NULL)    {
                if ((flist = fopen(FILELIST, "rt")) != NULL) {
                    /* ------- count the text files ---------- */
                    while (fread(path, sizeof path, 1, flist))
                        file_count++;
                    printf("\n%d files in the data base.",
                                file_count);
                    return;
```

```
            }
            fclose(text);
        }
        fclose(slots);
    }
    printf("\nCannot open Index Data Base");
    exit(1);
}

/* --------- close the index files ---------- */
void close_database(void)
{
    delete_tree();
    fclose(slots);
    fclose(text);
    fclose(flist);
}

/* --------- add a word to the index
       or add the file number to the existing word -------- */
void addindex(char *word, int fileno)
{
    long slotno;
    long hash;
    struct bitmap map1;
    long ptr = -1L;
    char wd[MAXWORDLEN+1];

    /* ------- compute a randon address from the word ------ */
    hash = compute_hash(word);
    hash *= sizeof(long);
    /* ------ read the random slot value -------- */
    fseek(slots, hash, SEEK_SET);
    fread(&slotno, sizeof(long), 1, slots);
    if (slotno == -1L)  {
        /* --- empty slot, add the word to the data base --- */
        fseek(text, 0L, SEEK_END);
        slotno = ftell(text);
        /* ----- set the bit map to this file only ----- */
        memset(&map1, 0, sizeof(struct bitmap));
        setbit(&map1, fileno);
        fwrite(&map1, sizeof(struct bitmap), 1, text);
        /* -- set the chain pointer to the terminal value -- */
        fwrite(&ptr, sizeof(long), 1, text);
        fwrite(word, strlen(word)+1, 1, text);
        /* --- insert the text address into the slot file --- */
        fseek(slots, hash, SEEK_SET);
        fwrite(&slotno, sizeof(long), 1, slots);
        return;
    }
    /* -------- the hashed slot is in use -------- */
    for (;;)    {
        /* --- point to the text index record ---- */
        fseek(text, slotno, SEEK_SET);
        fread(&map1, sizeof(struct bitmap), 1, text);
        fread(&ptr, sizeof(long), 1, text);
        fgets(wd, MAXWORDLEN+1, text);
        /* --- see if the entry matches the word --- */
        if (strcmp(wd, word) == 0)  {
            /* ---- the word matches this entry,
                    set this file's bit ---- */
            setbit(&map1, fileno);
            fseek(text, slotno, SEEK_SET);
            fwrite(&map1, sizeof(struct bitmap), 1, text);
            break;
        }
        /* ----- see if there is a chain ----- */
        if (ptr == -1)  {
            /* ------ end of the chain; add this word ---- */
            long newslotno;
            /* ---- to the end of the text index file ---- */
            fseek(text, 0L, SEEK_END);
            newslotno = ftell(text);
            /* ----- set the bit map to this file only ----- */
            memset(&map1, 0, sizeof(struct bitmap));
            setbit(&map1, fileno);
            fwrite(&map1, sizeof(struct bitmap), 1, text);
            fwrite(&ptr, sizeof(long), 1, text);
            fwrite(word, strlen(word)+1, 1, text);
            /* ----- chain the last entry to this one ----- */
            fseek(text, slotno+sizeof(struct bitmap), SEEK_SET);
            fwrite(&newslotno, sizeof(long), 1, text);
            break;
        }
        /* ------ the text record is chained
            (multiple words hash to the same slot) ---- */
        slotno = ptr;
    }
}

/* ------ search the data base for a match on a word ------ */
struct bitmap search_index(char *word)
{
    long slotno = -1;
    long hash;
    struct bitmap map1;
    char wd[MAXWORDLEN+1];

    /* ----- preset the bit map to all zeros ----- */
    memset(&map1, 0, sizeof(struct bitmap));
    /* ---- compute random slot address for the word ---- */
    hash = compute_hash(word);
    hash *= sizeof(long);
    fseek(slots, hash, SEEK_SET);
    fread(&slotno, sizeof(long), 1, slots);
    /* ------- navigate the chain -------- */
    while (slotno != -1)    {
        fseek(text, slotno, SEEK_SET);
        fread(&map1, sizeof(struct bitmap), 1, text);
        fread(&slotno, sizeof(long), 1, text);
        fgets(wd, MAXWORDLEN+1, text);
```

(continued on page 144)

Listing Five *(Listing continued, text begins on page 123.)*

```
        if (strcmp(wd, word) == 0)
            /* ---- the word matches this entry ---- */
            break;
        memset(&map1, 0, sizeof(struct bitmap));
    }
    return map1;
}

/* ---- compute a random address from an ASCII string ---- */
static unsigned compute_hash(char *word)
{
    unsigned hash = 0;
    while (*word)
        hash = (hash << 7) + (hash >> 9) + *word++;
    return hash;
}

/* ------ sets a designated bit in the bit map ------ */
static void setbit(struct bitmap *map1, int bit)
{
    int off = bit / 16;
    int mask = 1 << (bit % 16);
    map1->map[off] |= mask;
}

/* ------ tests a designated bit in the bit map ------ */
int getbit(struct bitmap *map1, int bit)
{
    int off = bit / 16;
    int mask = 1 << (bit % 16);
    return map1->map[off] & mask;
}

/* --------- add a file name to the list ----------- */
void indexing(char *filename)
{
    /* ----- add the file path to the index file list ------ */
    memset(path, '\0', sizeof path);
    strcpy(path, filename);
    fwrite(path, sizeof path, 1, flist);
    file_count++;
}

/* ---- return the file name associated with an offset ---- */
char *text_filename(int fileno)
{
    fseek(flist, (long) (fileno * sizeof path), SEEK_SET);
    fread(path, sizeof path, 1, flist);
    return path;
}
```

End Listing Five

Listing Six

```
/* ------------- bldindex.c --------------- */

#include <stdio.h>
#include <string.h>
#include "textsrch.h"

static void index_file(char *filename);

void main(int argc, char *argv[])
{
    open_database();
    printf("\nTextSrch: Building a TextSrch Index File");
    parse_cmdline(argc, argv, NULL, index_file);
    close_database();
}

static void index_file(char *filename)
{
    FILE *fp;
    char word[MAXWORDLEN+1];
    extern int file_count;
    int wdctr = 0;

    printf("\nIndexing %s", filename);
    indexing(filename);
    /* ---- open the object file ---- */
    if ((fp = fopen(filename, "rt")) == NULL)   {
        printf("\nError: No such file");
        return;
    }
    putchar('\n');
    /* ---- extract words and add them to the index ---- */
    while (!feof(fp))   {
        extract_word(fp, word);
        /* -------- search the noise word list -------- */
        if (srchtree(word) != 0)    {
            if ((++wdctr % 100) == 0)
                printf("\r%5d words", wdctr);
            /* ------- add the word to the index ------- */
            addindex(word, file_count-1);
        }
    }
    printf("\r%5d words", wdctr);
    fclose(fp);
}
```

End Listing Six

Listing Seven

```
/* ----------- text.c ------------ */

#include <stdio.h>
#include <ctype.h>
#include "textsrch.h"

#define isTEXT(c) (isalpha(c) || isdigit(c) || \
                   c=='+' || c=='#' || c=='_')

/* --------- extract a word from an input stream ----------- */
void extract_word(FILE *fp, char *s)
{
    int i, c;

    c = i = 0;
    while (!isTEXT(c))
        if ((c = fgetc(fp)) == EOF)
            break;
    while (isTEXT(c))   {
        if  (i++ < MAXWORDLEN)
            *s++ = tolower(c);
        if ((c = fgetc(fp)) == EOF)
            break;
    }
    *s = '\0';
}
```

End Listing Seven

Listing Eight

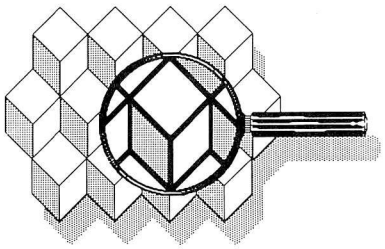
```
/* ---------- search.c ----------- */

/*
 * the TEXTSRCH retrieval process
 */

#include <stdio.h>
#include <string.h>
#include "textsrch.h"

/* ---- process the result of a query expression search ---- */
void process_result(struct bitmap map1)
{
    int i;
    extern int file_count;
    for (i = 0; i < file_count; i++)
        if (getbit(&map1, i))
            printf("\n%s", text_filename(i));
}

/* ------- search the data base for a word match -------- */
struct bitmap search(char *word)
{
    struct bitmap map1;

    memset(&map1, 0xff, sizeof (struct bitmap));
    if (srchtree(word) != 0)
        map1 = search_index(word);
    return map1;
}
```

End Listings

Listing One *(Text begins on page 134.)*

```
MODULE JTerm;

(* Stony Brook Modula-2 --  Last modified 10/31/89 *)
(* by Jeff Duntemann    --  For DDJ 2/90 *)

FROM CommPort IMPORT BaudRate, Parity, InitPort, StartReceiving,
                     StopReceiving, SendChar, GetChar, CommStatus;

FROM FileSystem IMPORT Close, File, Length, Lookup, Reset, SetPos, WriteChar;

FROM Keyboard IMPORT GetKey, CarriageReturn, LineFeed, AltX, F7, F8;

FROM Screen   IMPORT CreateWindow, CloseWindow, Color, DrawBorder,
                     MakeColorAttr, Position, SetCursor, Window,
                     WriteString, WriteLn, Xpos, Ypos;

FROM Terminal IMPORT Write, CharAvail, Read;

CONST
  Blink   = TRUE;
  NoBlink = FALSE;
  AsPopup = TRUE;
  NotAsPopup = FALSE;
  CreateFile = TRUE;

VAR
  CaptureOn   : BOOLEAN;
  CaptureFile : File;
  ItsLength   : LONGINT;
  Status      : CommStatus;
  Ch          : CHAR;
  OutString   : ARRAY[0..1] OF CHAR;
  Keystroke   : CARDINAL;
  W           : Window;
  WhiteOnBlue : CHAR;

BEGIN
  (* First set up the window to hold the terminal session: *)
  WhiteOnBlue := MakeColorAttr(White,Blue,NoBlink);
  W := CreateWindow(0,0,80,24,WhiteOnBlue,AsPopup);
  DrawBorder(W,MakeColorAttr(White,LightCyan,NoBlink));
  Position(W,1,0);
  WriteString(W,'\\\\JTERM\\\\  by Jeff Duntemann ',
              MakeColorAttr(Black,LightCyan,Blink));
  Position(W,1,1);
  SetCursor(W);

  (* Here we look for the capture file CAPTURE.TXT; open it if it  *)
  (* exists, and create it if it doesn't:  *)
  Lookup(CaptureFile,"CAPTURE.TXT",CreateFile);
  Length(CaptureFile,ItsLength);  (* Find out how long file is...  *)
  SetPos(CaptureFile,ItsLength);  (* ...and position it to EOF.    *)
  CaptureOn := FALSE;             (* Default to NOT capturing text *)

  (* Next, set up the interrupt-driven serial port and turn it on: *)
  Status := InitPort(0,Baud1200,7,1,Even);
  Status := StartReceiving(0,256);

  (* We check for keystrokes, then check for incoming data: *)
  LOOP  (* EXIT the loop (and the program) on Alt-X *)
    IF CharAvail() THEN
      GetKey(Keystroke);  (* Get a keystroke from the buffer *)
      CASE Keystroke OF
        CarriageReturn:   (* If CR was pressed, send CR *AND* LF: *)
          SendChar(0,CHR(CarriageReturn),FALSE);
          SendChar(0,CHR(LineFeed),FALSE); |
        F7: CaptureOn := TRUE;  |  (* F7/F8 toggle capture *)
        F8: CaptureOn := FALSE; |  (*  on and off *)
        AltX: EXIT;                (* Alt-X quits the program *)
        (* Send any non-command to the serial port: *)
        ELSE SendChar(0,CHR(Keystroke),FALSE);
      END;
    END;
    (* If a char's waiting in the comm buffer, get it and parse it: *)
    IF GetChar(0,Ch) = Success THEN
      OutString[0] := Ch;
      CASE ORD(Ch) OF
         (* This is how we fake TTY backspace: *)
        8: IF Xpos(W) > 0 THEN
             Position(W,Xpos(W)-1,Ypos(W));
             WriteString(W,' ',WhiteOnBlue);
             Position(W,Xpos(W)-1,Ypos(W));
             SetCursor(W);
           END;           |
       13: WriteLn(W);
           SetCursor(W);
           IF CaptureOn THEN
             WriteChar(CaptureFile,Ch)
           END;           |
       10: IF CaptureOn THEN
             WriteChar(CaptureFile,Ch)
           END;
       ELSE WriteString(W,Ch,WhiteOnBlue);
            SetCursor(W);
            IF CaptureOn THEN
              WriteChar(CaptureFile,Ch);
            END;
          END;
        END;
      END;
      (* Finally, we shut down the interrupt-driven input buffer: *)
      Status := StopReceiving(0);
      Close(CaptureFile);
    END JTerm.
```

End Listing



A set of object-oriented numerical tools that allows you to do complicated operations on vectors and matrices in the same manner as with built-in types has been announced by **Rogue Wave**, a Seattle outfit that specializes in object-oriented numerics. All of the standard C arithmetic operators such as "+" and "+=" have been extended to include vectors and matrices, and so have the C functions such as *cos( )* and *abs( )*. The set also includes new functions for statistics and numerical modeling applications, as well as a complex number class. Fast Fourier Transform server classes allow you to take the FFT or inverse FFT of any length series (real or complex). By using the inheritance property of C++, the company claims, new classes can be created from the Rogue Wave classes to do specialized tasks. The package includes a user's guide and reference manual. The cost is $150. Reader service no. 21.
Rogue Wave
P.O. Box 85341
Seattle, WA 98145-1341
206-523-5831

The C-scape Interface Management System and the Look & Feel Screen Designer have been announced by the **Oakland Group**. C-scape, Version 3.1, runs in either text or graphics mode, which means that C-scape menus, windows, and so on can appear on top of VGA graphics, and that graphics such as a PCX file image can appear inside a C-scape window. CGA, VGA, EGA, and most modes of Hercules are supported. C-scape is built upon the Oakland Windowing Library (OWL), which manages windows management, automatically tracks visible and hidden screen portions, supports as many windows as RAM can allow, and lets you write hidden windows.

The Look & Feel is new, and lets you design screens and run them in simulate mode, and then convert them to C code or save them to screen files that are callable at run time. All code is portable among DOS, OS/2, QNX, Xenix, Unix, and VMS.

C-scape supports DOS and OS/2 (in the same package), Microsoft and Borland C, and includes source code (for Unix and Xenix implementations, too). The Unix version supports color, and X Windows support should be available by the time this issue hits the streets. The Phar Lap DOS Extender and Rational Systems DOS/16M are also supported, as well as Zortech's C++ 2.0. The DOS version with the Look and Feel Screen Designer is $399, without $299. The Unix/Xenix version costs $999. Reader service no. 22.
Oakland Group, Inc.
675 Massachusetts Ave.
Cambridge, MA 02139-3309
800-233-3733

An MS-DOS advanced-overlay linker that supports CodeView, .RTLink/Plus, has been released by **Pocket Soft**. The nested overlay capability of this product supposedly allows the creation of substantially smaller memory images than are possible with Microsoft's linker overlay feature.

.RTLink/Plus also provides a link-time Profiler for performance analysis at user-adjustable timing intervals; the .RTL (run-time library) allows a group of .EXEs to share code from language libraries, third party libraries, and user's libraries, which prevents duplicate storage of common code; memory caching in extended or expanded memory for overlay swapping supposedly increases the speed of overlay handling on equipped systems; and Periscope, SoftIce, and MagicCV are supported. The list price is $495. .RTLink is identical to .RTLink/Plus, except that .RTLink does not provide the CodeView and Profiler features, and costs $295. Reader service no. 23.
Pocket Soft, Inc.
7676 Hillmont, Ste. 195
Houston, TX 77040
713-460-5600

The C Programmer's Toolbox for Apple's Macintosh Programmer's Workshop (MPW) has been announced by **MMC AD Systems**. The Toolbox has 20 new tools that complement the existing MPW tools. The list includes: CDecl, which translates and composes C declaration statements and cast operators; CFlow, which determines a program's function hierarchy and the composition of a program's run-time library, and includes an ANSI-compatible C preprocessor that allows inactive code sections to be ignored. CFlow also has the ability to process any number and size of C source files through a virtual memory system that uses temporary disk storage space when main memory is exhausted. CHilite highlights and prints C source files, as well as comments, keywords, Toolbox calls, function calls, macro calls, and definitions and/or user-defined symbols. And CLint checks for latent programming errors, including variable type usage, conditional and assignment statement usage, arithmetic operations in conditional expressions, and more. Some other tools are CPrint for reformatting, CXref for cross referencing, Cpp for preprocessing, Cat for concatenating input data streams, and CharCnt for counting input, characters, and so on.

The Toolbox works with any Macintosh with MPW 3.0 or later. MPW C is not required. The Toolbox works for C code generated by MPW C 2.x/3.x, Lightspeed C/Think C, Aztec C, all common PC C compilers, engineering workstations and Unix C compilers. It sells for $295. Reader service no. 24.
MMC AD Systems
Box 360845
Milpitas, CA 95035
408-263-0781

Version 3.0 of Star Sapphire Common Lisp is now being shipped by **Sapiens Software**. One of the few versions available for IBM and compatible PCs, Common Lisp is a general purpose, modern programming language, and the company claims it is "programmer friendly," encouraging greater productivity.

Star Sapphire Lisp supports the use of up to 8 Mbytes of extended or virtual memory, and includes a resident Emacs editor and online help for the whole language. The built-in incremental compiler and debugger supposedly speed program development. The product includes source code examples, including the Towers of Hanoi and the Colossal Cave Adventure, which demonstrate how to write programs in Lisp. An IBM PC, PS/2, or compatible with 640K memory and a hard disk are required. Version 3.0 is available for $99.95. Reader service no. 25.
Sapiens Software
P.O. Box 3365
Santa Cruz, CA 95063
408-458-1990

Two new control packages that are designed to make programming in Microsoft Windows and OS/2 Presentation Manager more efficient are Standard Control Pak and Tools Control Pak, by **Eikon Systems**. Each of these products offers three graphical, high-level, reusable software objects for use in the client area of windows or dialog boxes.

(continued on page 157)

(continued from page 148)

Jerry Weldon, one of Eikon's development engineers, claims that "high-quality graphical objects such as these can take weeks or months to refine and perfect, especially when you realize that each Control Pak consists of thousands of lines of source code."

In the Standard Control Pak, the Arrow window class can be used in place of the push buttons provided with Windows and OS/2 PM. The Palette window class provides a displayed set of predefined colors, and the Picture window class offers a variety of border styles for displaying bitmaps and metafile pictures.

The Tools Control Pak includes a Ruler window class for defining the position and spatial orientation of graphical objects, a Slider window class for selecting values from a range, and a Toolbox window class for selecting an operating mode from a rectangular array of small bitmaps, the limit of which is determined by available memory. The Standard Control Pak for Windows retails for $125 without source code, and $475 with; for OS/2 PM the cost is $225 without code, and $675 with code. The Tools Control Pak is $175 without source code, $525 with code. If a control is to be incorporated into any publicly distributed application, the source code must be purchased (but is royalty free). Reader service no. 26.

Eikon Systems, Inc.
989 E. Hillsdale Blvd., Ste 260
Foster City, CA 94404
415-349-4664

The **Video Electronics Standards Association (VESA)**, a group of hardware and software companies involved in computer graphics and displays, has adopted and published standards for Super VGA BIOS Extensions. This standard specifies a common software interface for Super VGA video adapters, providing a simplified means of accessing extended video modes such as 640 × 480 resolution in 256 colors. The standard follows the 800 × 600 16-color mode and 8514/A standards approved last year. Within the next couple of months, *DDJ* will publish an in-depth discussion of the VESA BIOS Extensions, along with related articles on 16-bit VGA and Super VGA. For copies of the standard, contact in writing (or FAX) the VESA office. Reader service no. 31.

VESA
1330 South Bascom, Ste. D
San Jose, CA 95128-4502
408-971-7525 (voice)
408-286-8988 (FAX)

A cooperative development effort to provide field-updatable BIOS capability for AT, MCA, and EISA-based personal computers has been announced by **Intel Corporation** and **Phoenix Technologies**. With Intel's 1-Mbit ETOX (EPROM tunnel oxide) flash memories and Phoenix's BIOS and utility software, the need to open the PC system case to swap BIOS chips will be eliminated. (Flash memory is a high-density, electrically erasable, nonvolatile semiconductor memory technology.)

Dick Pashley, general manager of Intel's Flash Memory Operation, claims that users "will be able to adapt their existing PCs to accommodate new peripherals requiring BIOS support, such as disk drives, screens, and keyboards." Intel's 28F010 1-Mbit flash memories have been available since late in 1989, and Phoenix's new BIOS updating utility program should be shipping by now. Both companies promise system design documentation and applications support. Reader service no. 27. Contact Intel for information.

Intel Corp.
Literature Dept. #6P19
3065 Bowers Ave., P.O. Box 58065
Santa Clara, CA 95052-8065

A structuring utility to transform Fortran IV and Fortran-77 into fully structured code (with or without VAX and Fortran-8x extensions) has been announced by **Cobalt Blue**. FOR_STRUCT v1.1 will replace Goto and If-Goto combinations with If-Then-Else, Do-While, and Do-Enddos, and offers style control switches to allow visual customization of the structured code. According to the company, FOR_STRUCT leaves original programming logic intact, does not duplicate code, and allows you to remove dead code segments during structuring.

FOR_STRUCT offers three levels of structuring: To Fortran-77; to Fortran-77 with VAX Do-While and Do-Enddo extensions; or to VAX Fortran-77 with Fortran-8X extensions. FOR_STRUCT is compatible with Cobalt Blue's FOR_C and FOR_C++ source code conversion packages. The Sun-3 and Sun-4 versions cost $1850, Xenix/Unix is $1450, and MS-DOS is $825. These prices include two months of free technical support and upgrades. Combined product discounts are also available. Reader service no. 28.

Cobalt Blue
2940 Union Ave., Ste. C
San Jose, CA 95124
408-723-0474

DDJ

(continued from page 160)

educational campaign. I'd like at this time to share this thought from one great American:

"I swore off computers for about a year and a half— the end of the ninth grade and all of the tenth. I tried to be normal, the best I could." — Bill Gates.

I know you are with me in praying that this fine young man gets back on the wagon.

For the technical community, the networks are going to continue running the thought-provoking "This is your CPU on bugs" message. But we're going to keep the message simple for the folks out there in the heartland: Bugs Are Bad.

In another aspect of the educational prong of this program, I am proposing the building of 8000 new jails nationwide. Never underestimate the educational value of six months in the slammer.

Enforcement: Get the bad guys. Education: Bugs Are Bad.

Third prong: Treatment. I will be presenting at SD90 a budget of $600 for bug treatment programs. Some people will say this is not enough. Some people think you can solve a problem by throwing money at it, but I think there are better things to throw.

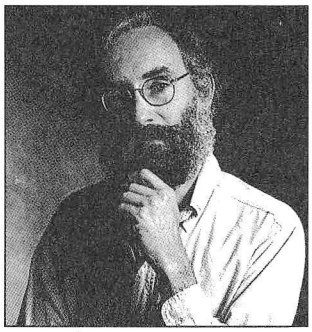
That's the three-pronged program. I'm pleased to announce that, to carry out this ambitious, comprehensive program, I have appointed my Cousin Corbett to the newly created post of Bug Czar. As head of the nation's bug eradication program, Corbett will have broad powers, including the right to search files without warrant, tap phones, disassemble code, and break shrink-wrap license agreements. I call this the CRime And Punishment, or CRAP, program. That's pronounced see-rap. One powerful tool that I am placing in Corbett's hands is the proposed new legislation supporting mandatory bug testing at all places of business.

I wish I could tell you that we're going to be able to eliminate all bug use in America, but that would be insincere. We will spare no expense to wipe out computer viruses, which are responsible for fully one-hundredth of one percent of lost productivity due to bugs. And you can rest assured that we will lean heavily on the cheapo shareware and freeware channels. I have the notorious Richard Stallman under FBI surveillance right now. But there will be a few common bugs that will not come under the jurisdiction of the BEA. We have no intention of harassing Microsoft, Ashton-Tate, Lotus, or any other producer of the fine software that has made this nation what it is today. After all, the occasional system crash is not really a bug. We will, however, continue to restrict use of their products on commercial air flights.

Michael Swaine

Michael Swaine
editor-at-large

The War on Bugs



The gravest domestic threat facing our nation today is software bugs.

Oh, I could talk about the deficit, the situation in Central America, the destruction of the tropical rain forests and the ozone layer, the homeless, the victims of last year's natural disasters, or this year's former employees of former Defense contractors. But what's the point? I'm not going to do anything about any of those problems. But when it comes to bugs, I have a program.

Bugs are our most serious problem in America today. Bugs are sapping our strength as a nation.

I'm holding in my hand, right now as I type this column, a bug-infected diskette, seized a few days ago by Bug Eradication Administration (BEA) agents in a park right across the street from the White House. Now, you may be wondering why a bug pusher would be doing his dirty work in such an unlikely place. I'll tell you why. He was there because I instructed BEA agents to lure him there and make their bust there so that I would have a dramatic story to tell you.

The point of the story is this: Bugs are bad. Bugs cost billions of dollars in lost productivity. Bugs threaten the strength of the family and the happiness of our children. Walk into any arcade in any large city in America and you'll see innocent children, red-eyed from crying because their high scores were erased by a software bug. It just breaks my heart to see that.

To combat this terrible menace, I have outlined and will be presenting at Software Development 90 this month a comprehensive program, based on the broad legislative and executive powers that you, the software development community, voted to give me at last year's SD89. I promised two things last year: No New Paradigms, and to make the streets safe for your sisters and daughters to walk. My program, which employs no new paradigms, is a street-cleaning, three-pronged attack in a national war on bugs.

Prong One: Enforcement. We've got to make it hot for the buglords. We know who they are: They're software developers. They are the people responsible for these terrible bugs invading our systems. Without them, there wouldn't be any bugs.

Too many people make excuses for these bug pushers. "Oh, that's not a bug," they say, "that's an undocumented feature." Or, "That's how the product is supposed to work; it's in the documentation." Well, those excuses aren't going to work any more. I will be recommending passage, by the attendees of the conference, of the Zero Tolerance Act for software bugs. Under this act, software written after April 1, 1990 must contain no bugs. To facilitate this plan, I will be presenting further legislation requiring that any software covered by the legislation and released after April 1, 1990, be written in Ada. This will, my advisors tell me, guarantee zero software bugs. This legislation calls for mandatory jail sentences for anyone programming in any other language than Ada. This will be implemented in an innovative six-and-six program: Six months on arrest, and another six months if convicted. If I've heard one thing from the law-enforcement community, it's that they're tired of having the courts release the criminals they arrest. This program should be a step in the right direction.

Forth programmers will be shot.

And I'm not forgetting the victims of the buglords: The users of buggy software. We're going to get them, too. I'm also presenting at SD90, as part of my comprehensive program, a far-reaching Victim Incarceration Program, or VIP, that will make it a criminal offense to knowingly use buggy software. That ought to bring the message home to everybody. The key here is, we all have to take responsibility for our actions.

I'm calling on all patriotic Americans to do their part. If you suspect someone in your workgroup of using buggy software, call this toll-free number: 1-800-BUG-ABOO.

And don't be deterred by the fact that the user is a member of your family. Sure, we can clean up the streets and the offices of America, but it won't do much good unless we clean up the families, too. Let's crush those bugs and put family values back into the family.

Prong Two: Education. I have outlined and will be presenting at SD90 a comprehensive, broad-bandwidth program of public bug education. The program is called BAB: Bugs Are Bad. I am gratified to announce that I have secured the cooperation of all the major television networks. This next season, all sitcoms, all miniseries, and all docudramas will carry the message to the American people: Bugs Are Bad. All network news programs will concentrate on software bug stories. And most important, major sports and entertainment figures have joined with me in this

(continued on page 158)

15 YEARS OF POWER PROGRAMMING!

#162 MARCH 1990 $2.95 (3.95 CANADA)

Dr. Dobb's JOURNAL

SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

ASSEMBLY LANGUAGE PROGRAMMING

Hand-Picked Hand Code

- ASM LIVES!
- 68040 PROGRAMMING
- MIXING ASM WITH HIGH-LEVEL LANGUAGES
- OBJECT-ORIENTED ASM

80386 DEBUGGING

MS WINDOWS MEMORY MANAGEMENT



0 38351 16562 8 03

CONTENTS

MARCH 1990
VOLUME 15, ISSUE 3

FEATURES

ASSEMBLY LANGUAGE LIVES! 16
by Michael Abrash
Assembly language isn't the be-all and end-all of PC programming, but as Michael states, it's sometimes the only game in town when performance or program size are important.

ASSEMBLY LANGUAGE TRICKS OF THE TRADE 30
by Tim Paterson
Every programmer collects a personal bag of programming tricks. Tim's has been 13 years in the making, and he shares some of his favorites with you.

68040 PROGRAMMING 38
by Stephen Satchell
The newest member of the 680x0 family provides some challenges for programmers at all levels, particularly when it comes to caching.

HOMEGROWN DEBUGGING — 386 STYLE! 46
by Al Williams
Use the 80386's hardware to debug your programs by including Al's assembly language code to establish breakpoints.

MANAGING MULTIPLE DATA SEGMENTS UNDER MICROSOFT WINDOWS: PART II 58
by Tim Paterson and Steve Flenniken
Last month, Tim and Steve presented a method for managing multiple data segments under MS Windows using the *segment* table. This month, they provide a sample Windows program that puts the *segtable* library to work.

OBJECT-ORIENTED PROGRAMMING WITH ASSEMBLY LANGUAGE 66
by Randall Hyde
Randy makes a case that the object-oriented paradigm isn't completely the domain of high-level programming languages. He believes that OOP techniques can be applied, and are worth considering for ASM projects too.

EXAMINING ROOM

INSIDE WATCOM C 7.0/386 74
by Andrew Schulman
Andrew suspects that Watcom's C 7.0/386 has launched the opening salvos in a 32-bit 386 development tool war. He also looks at how Novell has implemented the compiler for its C Network Compiler/386.

PROGRAMMER'S WORKBENCH

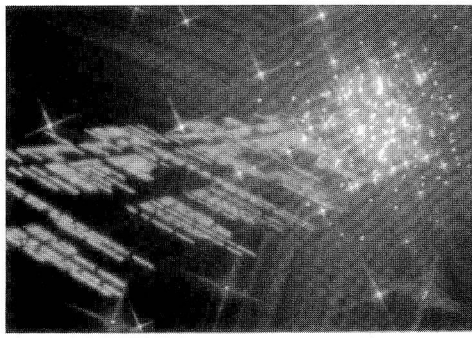
MIXED-LANGUAGE PROGRAMMING WITH ASM 84
by Karl Wright and Rick Schell
As Karl and Rick point out, it's not only practical but often advisable to mix languages and memory models in order to achieve the best results. Assembly language is a vital part of this mix.

COLUMNS

PROGRAMMING PARADIGMS 122
by Michael Swaine
Lisp has been codified, gentrified, and now objectified. Michael looks at how the Common Lisp data-type system underlies the object system, and how Lisp functions have been extended to the object world.

C PROGRAMMING 127
by Al Stevens
TEXTSRCH, Al's text retrieval project, continues to grow. Now you can select and view one of the files from within the TEXTSRCH program itself. He then uses this feature to explore the CURSES function library.

STRUCTURED PROGRAMMING 134
by Jeff Duntemann
There really were some neat ideas at last fall's Comdex, you just had to search them out. Jeff describes the jewels he discovered, then delves into sets in Modula-2.



DEPARTMENTS

EDITORIAL 6
by Jonathan Erickson

LETTERS 8
by you

SWAINE'S FLAMES 160
by Michael Swaine

PROGRAMMER'S SERVICES

OF INTEREST 152
compiled *by Janna Custer*

ADVERTISER INDEX 153
where to go for more information on products

PROGRAMMER'S MARKETPLACE 154
classified ads

NEXT ISSUE

If you've been scratching your head over neural networks, you'll want to pay special attention to our April issue. We'll also provide the long-awaited code implementation for our Rhealstone real-time benchmark, begin an in-depth examination of VGA, and include *DDJ's* 1989 index.

Patent Letter Suits

Mark Nelson's article on the LZW data compression algorithm (*DDJ* October, 1989) sparked a forest of fires, at least in respect to patenting algorithms. The first spark, if you recall, was a letter from Ray Gardner, pointing out that the LZW algorithm was patented by Unisys back in 1985 (see "Letters," December 1989). Mark's response answered a few questions but raised several more.

About the time we published Ray's letter and Mark's reply, the U.S. Court of Appeals settled a dispute between the U.S. Patent Office and Sharp Corporation in a case that revolved around Sharp's patent application for a voice-recognition circuit. The Patent Office had rejected Sharp's original application in part because they felt the circuit's only purpose was to execute an algorithm. And, the Patent Office insisted, algorithms can't be patented because they are nothing more than mathematical abstractions. Furthermore, the Patent Office felt that Sharp was trying to patent every possible means of implementing the algorithm, not just the way it was used in this particular voice-recognition circuit.

As it turned out, the Court of Appeals didn't agree with the Patent Office. The court said that an algorithm can be safeguarded, at least as how it is used to describe a physical device (like a circuit) or in terms of other functional equivalents of that algorithm.

To better come to grips with this issue, I called Charles Gorenstein, the Falls Church, Virginia attorney who represented Sharp. Early in our conversation, Mr. Gorenstein stated that "a purely mathematical algorithm is probably not patentable" but, he added, the specific methods of implementing an algorithm are patentable. In other words, what is patentable is the method, not the math. If someone developed a different circuit to execute Sharp's voice-recognition algorithm, that's fine and dandy. And apparently that's part of the basis of the Court of Appeal's decision.

Key to any patent grant is the concept of "new and unobvious," an area that Mr. Gorenstein feels the Patent Office has overlooked. Using a 1979 patent for spreadsheets as an example, he explained that just about anyone with a ledger, a pencil, and some data would fill out the rows and columns in much the same way as they would with an electronic spreadsheet. A ledger — and a spreadsheet — is obvious. He therefore questions whether the spreadsheet patent should ever have been granted. This question of "obvious" raises another important issue. What may be *unobvious* to those in the Patent Office may very well be obvious to technically sophisticated programmers like *DDJ* readers.

What all this leads up to is a letter I received from Bob Bramson, the Unisys patent attorney Mark mentioned in his response. I won't give a blow-by-blow account of the letter, you can read it for yourself on page 8, the first entry in this month's "Letters" section.

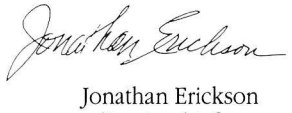
I will say that the letter is a politely worded clarification of Unisys's patent on the LZW algorithm, with only a slight sense of the steel behind it, at least in reference to Unisys's intention of going after infringers.

I'm sorry, but I still don't understand. It seems that if, as I think the court ruled, you can use Sharp's algorithm to design a different voice-recognition circuit, you should be able to use Sharp's (or Unisys's or anyone else's) algorithm for an entirely different purpose than it is used in the original patent. That is, you should be able to use the LZW algorithm in a program that has nothing to do with telecommunications or modems. This assumes, of course, that Unisys's patent is for the modem and the algorithm as it helps define the modem. I agree with Mark. Unisys will indeed be very busy tracking down programmers who have implemented some form of the LZW algorithm.

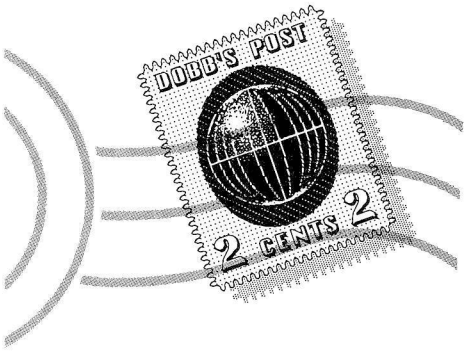
I'm all for any company, large or small, taking steps to protect R&D investments that give it a competitive edge. But it's distasteful for large companies to threaten smaller outfits with litigation that can't be won in the courts, but can be outlasted by a large company with the resources to do so. Now I'm not in any way suggesting this is Unisys's ploy, nor does Mr. Bramson even hint at this, it's far too often the way the world works.

In his response to the letter that started all of this, Mark suggested that software developers who intend on using patented algorithms (like LZW) in commercial products get some legal advice before proceeding. Mr. Gorenstein seconded this, even to the point of suggesting that programmers do a patent search prior to implementation. While this advice is sound and safe, it is also lengthy and expensive, luxuries that software developers usually can't enjoy.

Today's mail didn't bring a letter from a lawyer, but it did include a letter from Dan Abelow, a Newton, Massachusetts reader who specializes in analyzing emerging technologies, and who, coincidentally, proposes to write an article on "Enabling Patents." He calls the topic a "blossoming controversy [that] has failed to germinate positive suggestions" and, from what I can tell, he's making a case that software patents may actually encourage innovation and invention. I don't know that I agree with him, but I'm curious enough to give him a call and find out what he has in mind.



Jonathan Erickson
editor-in-chief



Patented Algorithms

Dear *DDJ*,

In the "Letters" column of your December 1989 issue, Mark Nelson discusses U.S. Patent 4,558,302 entitled "High Speed Data Compression and Decompression Apparatus and Method." This patent was developed by Terry Welch, a former Unisys employee, and is owned by Unisys. According to Mr. Nelson, I have been quoted as saying that Unisys will "license the algorithm for a one time fee of $20,000." As a concession to the modem industry, Unisys has agreed to license the patent to modem manufacturers for use in modems conforming to the V42.bis data compression standard promulgated by CCITT, for a one-time fee of $20,000. This $20,000 license, however, is not a general license under all applications of our patent but is limited to the specific application discussed above.

Responding to the second paragraph of Nelson's remarks, Unisys is actively looking into the possibility that a large number of software developers may be infringing one or more of our data compression patents. We have only recently become aware of these potential infringers and the process of taking action will take some time.

Unisys is happy to accept inquiries from persons interested in acquiring a license to U.S. Patent 4,558,302. If your readers have any further questions, they should contact Mr. Edmund Chung of our licensing office, at 313-972-7114.

Robert S. Bramson
Unisys
Blue Bell, Penn.

Say It Ain't So

Dear *DDJ*,

Dan W. Crockett's assertion in the January 1990 *DDJ* "Letters" section that structured programming requires that each functional node (or implementation unit) have only a single parent is alarming, and damned difficult to program in the real world. I think that he interprets the abstract requirements of structured programming a little too literally when it comes to coding.

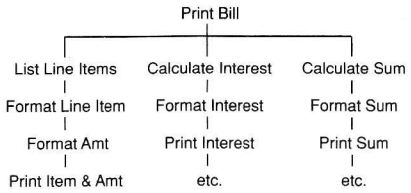
As an example, consider a Pascal function, which formats dollar amounts for output. The function might take a real dollar argument and translate it to a "*$nnn,nnn, . . . ,nnn.nn*" format, and be declared as function *DollarFmt(v : real):string;* The whole point of having the function is that it *can* be called from any procedure or function in a program; if the dollar amounts are formatted incorrectly we can first check to see if the error lies in *DollarFmt*, because it is solely responsible for performing the task.

This is structured programming: Breaking down a task into smaller and smaller (and finally, logically indivisible) subtasks. Subtasks which perform similar or identical tasks can then be coded as a single (probably parameterized) routine.

Mr. Crockett wants program structure to be a B, Quad, or whatever tree, which is fine, but reality demands that the implementation be a threaded tree. Under the Crockett scheme we would be forced to write a separate *DollarFmt* for each caller (*AmountDue_DollarFmt, AmountPaid_DollarFmt, ad nauseam*)! The resulting plethora of duplicate routines would produce a worse debugging situation than Mr. Crockett thinks he'll have already — never mind the maintenance nightmare.

The "single" restriction structured programming is the requirement that a single functional node not have more than one entry point *within it*; which is to say that all callers must enter through the same door. It is perfectly reasonable for a routine to have more than one caller — without multiple callers there would be little reason for building a distinct procedure or function for performing the task.

Going back to the *DollarFmt* example: The structural decomposition of a hypothetical bill printing task might be



It is (hopefully) obvious that "Format Amt," "Format Interest,"and "Format Sum" should be programmed as calls to a single formatting routine, even though they are different tasks in the abstract.

There are dangers in interpreting any abstraction too literally. And there is that other thing, in the word of Will Rogers: "It's not what we don't know that hurts, it's what we know that ain't so."

Brook Monroe
Durham, North Carolina

Locator Fix

Dear *DDJ*,

The listing of Mark Nelson's "Locate tool" in the January 1990 issue has a bug in the *read_header_data* procedure: It occurs in his calculation of *image_size*. The line:

```
image_size = (header.file_
             size_in_pages - 1) * 512;
```

should be replaced with:

```
if (header.image_length_mod_512
                              = = 0)
   image_size = header.file_
                size_in_ pages * 512;
else
   image_size = (header.file_
             size_in_pages - 1) * 512;
```

The bug occurs when the actual image size is an even multiple of 512 bytes. As an example, consider an image size of 1526 (512 * 3). In this case, *header_ file_size_in_ pages* would be three and *header.image_length_mod_ 512* would be zero. Mark's code would produce an incorrect size of 1024 due to the decrement of *header.file_size_ in_ pages.*

I had the opportunity of stumbling into this when writing a combination .EXE loader/relocator/unrelocator for a non-DOS-based embedded control system.

Thank you for your time and keep the interesting articles like Mark's coming.

Bill Trutor
Holden, Mass.

Mark responds: Bill has correctly identified a mistake in my program. I think I might have avoided this mental error with better naming of structure members. In any case, this is one of those program errors that occurs so infrequently (1 out of 512 links) that it can be extremely elusive, so thanks to Bill for pointing it out.

Data Structure Dream Machine

Dear *DDJ*,

In Jeff Duntemann's column in the December 1989 issue of *DDJ*, he mentioned his dream system under Windows 386. I have a question about this. I understand the languages and the PageMaker part, but could he expand

(continued on page 12)

(continued from page 8)
on using Paradox? Do I understand him to mean that you use it to keep track of details about your data structures? Sounds interesting; could he elaborate?

Guy Townsend
CIS 73040,1671

Jeff's response: Hate to be a spoilsport, but mostly what I use Paradox for is to keep my various contact files a keystroke away. The notion of using a real-relational database to manage the gritty details of major development projects is a good one, but the language vendors are going to have to do the integration between the tools and the database. Some major vendors are indeed working on this, (still secretly) and you'll be seeing the results in DDJ *when they surface.*

Intek Heard From

Dear *DDJ*,

From time to time you must hear from disgruntled companies who feel that they have suffered at the hands of one of your writers performing a post-mortem with an axe.

Knowing, however, that Al Stevens is a venerable pilgrim to the hallowed halls of Bell Labs and a proponent of the object-oriented paradigm, we suppose that his summarial execution of our product was caused by a bit of underdone potato.

In his November "C Programming" column, just after explaining to his readership that he was neither rigorous nor controlled, he set about to describe the available C++ compilers and translators available for DOS. Without rigor or control he dismissed our Intek C++ product without evaluating the product at all! He chose instead to fault an example program that we provide with our distribution of the AT&T translator. This example program (which we supply the source to in the product distribution) invokes the C preprocessor, the C++ translator, and the target C compiler in succession. We supplied it to provide our customers with a convenient method of progressing from source to executable if they were invoking from the command line. The mentioned bug occurred only with DOS 3.3, and as with all software companies that stand behind their product with integrity, we supplied the fix to all of our customers long before Al's column went to print. Of course, Al didn't mention that other translator products don't offer anything like this and certainly don't supply the source to such a program.

Did Al mention that the near, far, huge, pascal, fortran, and cdecl keywords don't work with the AT&T distribution or with some other translator products but that they do with Intek C++? No. Ever tried to use a third party header file with some of these keywords in it or try to link to a library, expecting the results of these modifiers, Al?

Did Al mention that if one tried to compile any production size C++ source modules with other products that they would run out of memory? No. Maybe he'd rather make sure that all of his source files were less than 4K in size and that he could only include three or less header files.

Did Al benchmark the fact that the only product from among the group that he mentioned that will compile the AT&T C++ source distribution is Intek C++? No. (That's AT&T's definition of the robustness of a C++ translator implementation by the way.)

Please assure Al that Intek C++ will continue to have a future in the PC world. Our client list has many of the Fortune 500 firms among it. We also use our own product in providing factory automation applications to the major workstation and computer manufacturers in the country.

We feel like you would feel if in a review of magazines *Dr. Dobb's* was dismissed as not being a quality software tools magazine because it sounds like it should be a medical journal.

Mac Cutchins
Intek
Bellevue, Wash.

Al responds: My evaluation of Intek C++ consisted at first of the seemingly simple task of getting it to compile the hello.cpp program that comes with the translator. "Hello, world," nothing more, right out of the box. That simple task involved two days of frustration and several phone calls to Intek.

The Intek technical support person at first insisted that there was something wrong with my setup. The nature of the bug — the translator worked every other time — encouraged both of us to believe that. The compiler failed, I called, he made a suggestion, the compiler worked. I hung up, the compiler failed, I called, and so on. One of those times we changed operating systems, and the technical support person concluded that my copy of DOS 3.3 was the culprit. He must have remembered that episode and subsequently convinced you, Mr. Cutchins, but not me. The bug was identical for DOS Versions 3.0, 3.2, 3.3, and 4.0. Under 3.1, the bug is different, and hello.cpp just never compiles. When I reported these findings, your technical support person, by now tired of hearing from me, curtly announced that there must be some problem, that it would get fixed, goodbye, and thank you very much. If you fixed that bug, I never heard about it, before or after my column went to print. Until now, that is. I guess as a pesky magazine columnist with a free review copy of your pricey product, I don't rate an upgrade. Never once during all those calls did Intek suggest that I abandon the "example" CPLUS program and use the lower-level programs, which I now see is the obvious solution.

In my opinion, Intek C++ is underpackaged and over-priced. The skimpy 40-page spiral-bound manual devotes only 10 pages to installing and using the translator, has exactly two sentences about using it with Turbo C, has some critical typos (the C_COMPILER environment variable is misspelled, for example), and never lets on that the CPLUS program is a mere example to be used at one's own risk. Intek C++ fails to measure up to the standards of quality that PC programmers have come to expect in their language products. My assessment of your future in the PC market was based on my view of the cost and quality of the Intek C++ software, documentation, and support and of the expensive hardware/software foundation necessary to use it. I stand by that assessment. If you believe that Intek C++ has improved substantially since my evaluation of it, I'll be pleased to give it another look.

Round and Round We Go

Dear *DDJ*,

Recently I completed a graphics course, so I read with interest the January issue article by Robert Zigon dealing with generation of circles. I found the article to be a clear and well written exposition of the problem. However, any algorithm based upon the parametric representation of a circle must involve significant overhead in the form of floating-point calculations. A superior algorithm developed by J.E. Bresenham some years ago avoids such overhead.

The Bresenham algorithm makes use of the fact that screen coordinates are integer valued, so it should be possible to select the circle's coordinates using only integer arithmetic as well. Use of only integer arithmetic is the key to the efficiency of the algorithm. The algorithm is used to advance along the perimeter of the circle, selecting the adjacent pixel which is nearest to the circle at each step. Because of circular symmetry, it suffices to determine only one-eighth of the circle using this technique.

An excellent derivation of the algo-

(continued on page 14)

Dr. Dobb's JOURNAL
SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

PUBLISHER *Peter Hutchinson*

EDITORIAL
EDITOR-IN-CHIEF *Jonathan Erickson*
MANAGING EDITOR *Monica E. Berg*
TECHNICAL EDITORS *Michael Floyd, Ray Valdes*
EDITORIAL ASSISTANT *Janna Custer*
CONTRIBUTING EDITORS *Al Stevens, Jeff Duntemann, Martin Tracy, David Betz, Tom Genereaux, Andrew Schulman*
COPY EDITORS *Rhoda Simmons, Pamela Dillebay, Nan Fornal*
EDITOR-AT-LARGE *Michael Swaine*

ART/PRODUCTION
ART/PRODUCTION DIRECTOR *Larry L. Clay*
ART DIRECTOR *Michael Hollister*
PRODUCTION SUPERVISOR *Amy Shulman Lesovoy*
TECHNICAL ILLUSTRATOR *Linda Ann Clark*
TYPOGRAPHERS *Teresa Raines, Margaret Anderson, Charlene Carpentier*
COVER PHOTOGRAPHER *Michael Carr*

CIRCULATION
DIRECTOR OF CIRCULATION *Maureen Kaminski*
CIRCULATION MANAGER *Randy Robertson*
CIRCULATION PLANNING MANAGER *Manny Sawit*
DIRECT MARKETING MANAGER *Andrea Weingart*
NEWSSTAND MANAGER *Sarah Forsman*
DIRECT MARKETING COORDINATOR *Francesca Davies*
PROMOTION COORDINATOR *Pam Moore*
FULFILLMENT COORDINATOR *Anne Jean*

ADMINISTRATION
VICE PRESIDENT OF FINANCE *Kate Deschamps*
CONTROLLER *Mary Collopy*
CREDIT MANAGER *Betty Arsene*
ACCOUNTING SUPERVISOR *Renate Kernke*
ACCOUNTS RECEIVABLE *Wendy Ho*
ACCOUNTS PAYABLE *LuAnn Rocklewitz*

MARKETING/ADVERTISING
DIRECTOR OF SALES AND MARKETING *Karla Spormann*
ADVERTISING COORDINATOR *Laura Stack Pullen*
MARKETING ASSISTANT *Sara Noah Ruddy*
ACCOUNT MANAGERS *see page 152*

M&T PUBLISHING INC.
CHAIRMAN OF THE BOARD *Otmar Weber*
DIRECTOR *C. F. von Quadt*
PRESIDENT *Laird Foshay*
VICE PRESIDENT OF PUBLISHING *William P. Howard*
VICE PRESIDENT/GROUP PUBLISHER *Randall L. Stickrod*

DR. DOBB'S JOURNAL (USPS 307690) is published monthly, except semimonthly in December, by M&T Publishing, Inc., 501 Galveston Dr., Redwood City, CA 94063; 415-366-3600. Second-class postage paid at Redwood City and at additional entry points.

ARTICLE SUBMISSIONS: Send manuscripts and disk (with article and listings) to the editorial assistant 415-366-3600.

***DDJ* ON COMPUSERVE:** Type GO DDJ.

***DDJ* LISTING SERVICE:** 603-882-1599. Supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Type *listings* (use lowercase) at the login prompt.

SUBSCRIPTION: $29.97 for 1 year; $56.97 for 2 years. Foreign orders must be prepaid, including the additional postage (air or surface) in U.S. funds drawn on a U.S. bank. Add $13 for surface mail to all addresses out of the U.S.; add $33 for airmail to Canada and Mexico; or $32 for airlift to all other countries.

POSTMASTER: Send address changes to *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. **ISSN 1044-789X**

CUSTOMER SERVICE: For subscription orders, questions, and changes of address call toll-free 800-456-1215 (U.S. and Canada) or write *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. For book/software orders call 800-533-4372 (in California 800-356-2002).

FOREIGN NEWSSTAND DISTRIBUTOR: Worldwide Media Service Inc., 115 E. 23rd St., New York, New York 10010; 212-420-0588 FAX 212-420-1265.

Entire contents copyright ©1990 by M&T Publishing, Inc., unless otherwise noted on specific articles. All rights reserved.

The Audit Bureau

(continued from page 12)

rithm is given in the text *Computer Graphics* by Donald Hearn and M. Pauline Baker (Prentice Hall, 1986). The derivation depends only upon elementary algebra, but may require somewhat greater mathematical maturity due to the notation used. The text also presents Pascal code for the algorithm. Another reference, which gives a limited explanation and a C code implementation of the algorithm, but which does not attempt to derive the algorithm, is *Graphics Programming in C* by Roger T. Stevens (M&T Books, 1988).

Joseph M. Hovanes Jr.
Pittsburgh, Pennsylvania

Forth Fan

Dear *DDJ*,

Here's 20 cents to fan the Flames of T.S. Kuhn's book, *The Structure of Scientific Revolutions*. It caused me to go cold turkey re. the tube for three days.

Martin Tracy reaffirmed my belief that Forth in its dialects offers the best presently available forum for discussion of "discrete mathematics" and the foundations of computing science. But I would like to see his work in the form of a bootable operating system and not a guest under another commercial product.

I confess that my own present work, "simpli-Forth," which is strongly tied to the 6502, still requires a fig-Forth boot to get off the ground. Perhaps if I work, I can learn enough about target compilers to create my own boot codes.

It seems to me that small operating systems with too-early emphasis on hiding or transportability may not be in the best interest of learners who seek to know in detail how their computing systems work. I would like to see small Forth systems place the user in a programming environment which makes plain the processes of his machine. That is why calls to DOS seem misplaced to me; I'd prefer that all of a small Forth system be available to the decompiler and user.

Would not a system for the programming of "smart" peripherals be more useful and general by omission of read-only memory? One could imagine modified error-handling, perhaps by redirecting the error-message stream to the calling device and transmitting a raise-error-request to it. But I remain convinced that the "smart" external should be executing a standard and expandable Forth kernel, albeit a minimal one, and that communication with it should be in the form of a standard, interpreted input stream.

The user of such a device could then load codes indicating how the forthcoming data is to be handled, followed by the commands to be executed and the data (e.g., 80 PRINTLINE THE QUICK BROWN FOX JUMPS OVER . . .). At the end of such a session, some command such as DONE would then forget the loaded object-behavior back to that formerly executing. Instead of relying on ROM to make our machines robust we would enter a new arena of opportunity for flexibility. It is time for a generation of peripherals which can follow the lead and dance.

On another subject, Brodie encourages us, "Use dumb words." One of the major differences between fig-Forth and Forth-83 is in the use of the STATE variable and its effect on words such as ' and LITERAL. In the process of learning to use STATE-sensitive words correctly, I, too, have been hopelessly confused from time to time. But the fully interactive capabilities possible in a modern Forth machine may require STATE-sensitive behavior.

For this reason I chose to write SIF (STATE @ IF) which may be used:

```
: TEST SIF COMPILE THEN DO-IT ;
                              IMMEDIATE
```

which will cause TEST to compile DO-IT if compiling else execute it (COMPILE is not IMMEDIATE). Although this example makes TEST equal to DO-IT, more-useful examples can be drawn. Another word might be ?COMPILE that would combine the effects of SIF, THEN, and IMMEDIATE and be used: : TEST ?COMPILE DO-IT ; so that all words using ?COMPILE would automatically be made IMMEDIATE.

"Use dumb words" is sound advice. But some quite interesting capabilities arise only when one uses correctly written words with STATE-sensitive behavior.

Jon W. Osterlund
Greeley, Colo.

DDJ

We welcome your comments (and suggestions). Mail your letters (include disk if your letter is lengthy or contains code) to DDJ, 501 Galveston Dr., Redwood City, CA 94063, or send them electronically to CompuServe 76704,50 or via MCI Mail, c/o DDJ. *Please include your name, city, and state. We reserve the right to edit letters.*

Assembly Language Lives!

More Speed, Less Filling

Michael Abrash

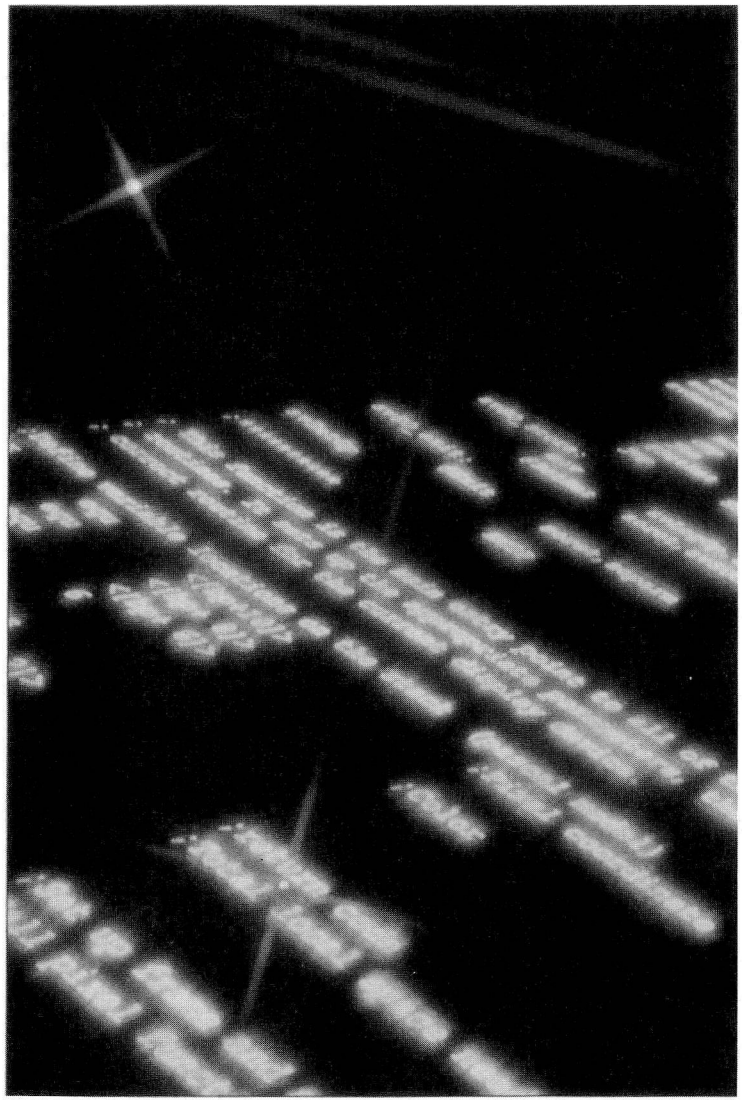
There's an old joke that goes something like this:

Person #1: Help! My brother thinks he's a chicken, and I don't know what I should do.
Person #2: Have you told him the truth?
Person #1: I would, but I need the eggs.

Updated for the modern age of structured languages and object-oriented programming, that joke would read:

Manager #1: Help! My programmers think assembly language is a viable programming language, and I don't know what I should do.
Manager #2: Have you told them the truth?
Manager #1: I would, but I need the speed.

Assembly language beats everything else hands down when it comes to performance — especially when programming for the 80x86, where assembly language is wild, woolly, and wondrous — yet it gets no respect. When you flat-out need performance, there simply are no substitutes for assembly language — so why doesn't anyone seem to love it?



Assembly Language Isn't Cheap

Experts, pundits, and management types have been beating the drums for the demise of assembly language for years. There are many good reasons for wishing it dead. Compared to compiled code, good assembly-language code is harder to write, is more bug prone, takes more time to create, is harder to maintain, is harder to port to other platforms, and is more difficult to use for complex, multiprogrammer projects. That makes assembly language an expensive, demanding, and time-consuming development language. Given the realities of time to market, the relative costs of good assembly language and high-level language programmers, programmer turnover, and ever-increasing

Michael works on high-performance graphics software at Metagraphics in Scotts Valley, Calif. He is also the author of Zen of Assembly Language *published by Scott, Foresman & Co., and* Power Graphics Programming, *from Que.*

software complexity, it's neither surprising nor unreasonable that most of the industry wishes assembly language would go away.

Assembly language lives, though, for one simple reason: Properly applied, it produces the best code of any language. By far.

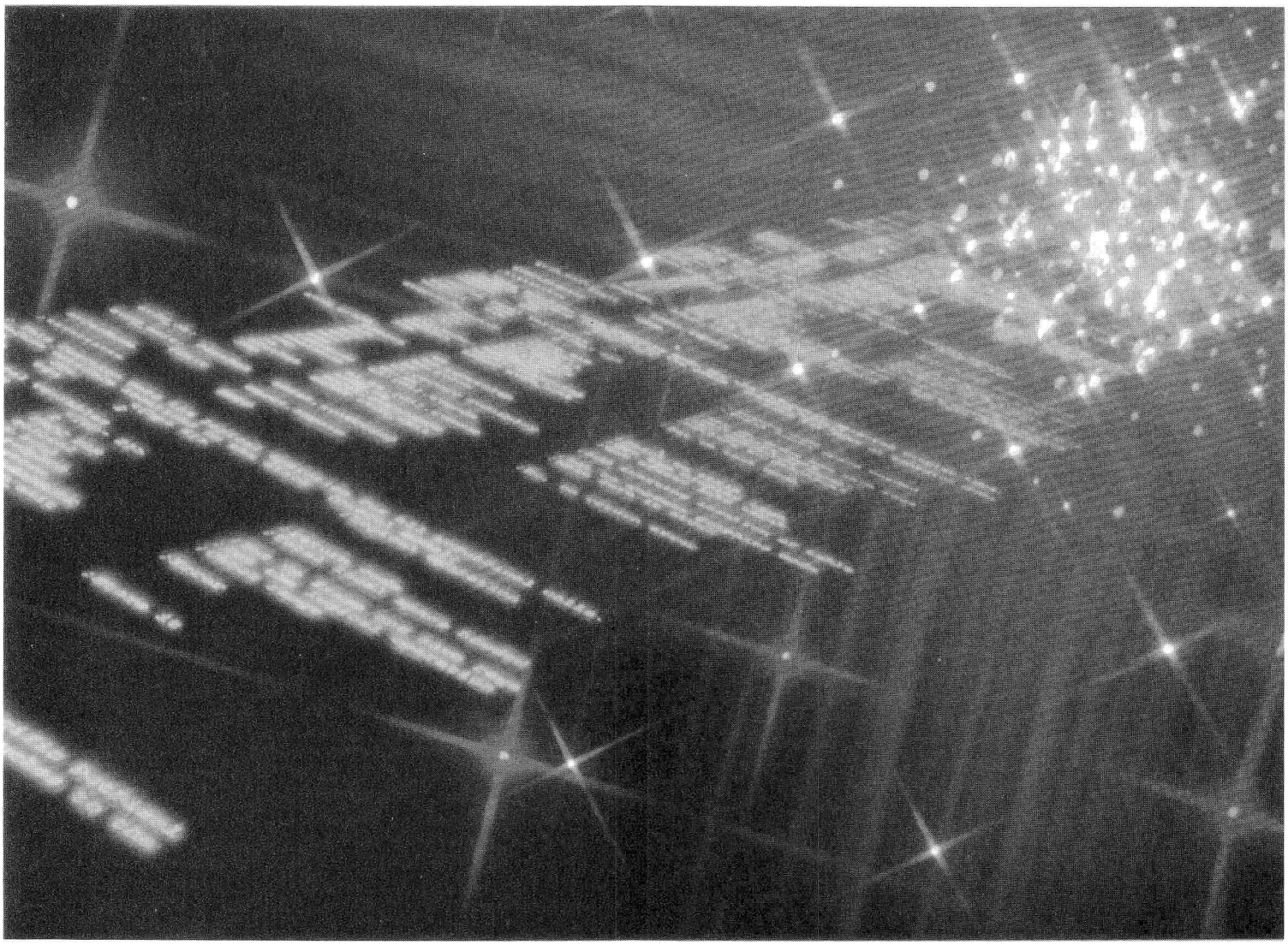
Assembly Language Lives

Don't believe me? Consider this. If the carbon-based computer between your ears were programmed with as good a compiler as Microsoft's, then you'd generate much better code in assembly language than does Microsoft C, because you know vastly more about what you want your program to do and are marvelously effective at integrating that knowledge into a working whole. High-level languages are artificially constrained programming environments, able to pass relatively little of what you know along to the ultimate machine code. There are good reasons for that: High-level languages have to be compilable and comprehensible by humans. Nonetheless, there's no way for a high-level language to know where to focus its efforts, or which way to bias code.

For example, how can a Pascal compiler know that one loop repeats twice, on average, while another repeats 32,767 times? How can a C compiler know that one subroutine is time critical, deserving of all possible optimization, while another subroutine executes in the background while waiting for the next key to be pressed, so speed matters not at all? The answer is: No way. (Actually, *#pragma* can do a little of that, but it's no more than a tiny step in the right direction.)

Just as significantly, no compiler can globally organize your data structures and the code that manipulates those structures to maximum advantage, nor take advantage of the vast number of potential optimizations as flexibly as you can. (Space forbids even a partial listing of optimization techniques for the 80x86 family: The list is astonishingly long and varied. See Tim Paterson's article in this issue for a small but potent sample.) When it comes to integrating all the information about a particular aspect of a program and implementing the code as efficiently as possible given the capabilities of a particular processor, it's not even close: Humans are much better optimizers than compilers are.

Almost any processor can benefit from hand-tuned assembly language, but assembly language lives most vibrantly in the 80x86 family. The 80x86 instruction set is irregular; the register set is small, with most registers dedicated to specific purposes; segments complicate everything; and the prefetching nature of the 80x86 renders actual execution time non-quantifiable — and optimization at best an art and at worst black magic — making the 80x86 family a nightmare for optimizing-compiler writers. The quirky



(and highly assembly language amenable) instructions of the 8086 live on in the latest 80x86-family processors, the 80386 and 80486, and will undoubtedly do the same for many generations to come. Other processors may lend themselves better to compilers, but the 80x86 family is and always will be a wonderland for assembly language.

Consider this: Well-written assembly language provides a 50 to 300 percent boost in performance over compiled code (more sometimes, less others, but that's a conservative range). An 8-MHz AT is about three times faster than a PC, a 16-MHz 80386 machine is about twice as fast as an AT, and a 25-MHz 80386 is about three times as fast as an AT. There are a lot of PCs and ATs out there — 20 to 30 million, I'd guess — and there is a horde of users contemplating the expenditure of thousands of dollars to upgrade.

When you absolutely, positively need to keep program size to a minimum, assembly language is the way to go

Now consider this. Those users don't have to upgrade — they just need to buy better-written software. The performance boost good assembly language provides is about the same as stepping up to the next hardware platform, but the assembly language route is one heck of a lot cheaper.

In other words, better software can eliminate the need for expensive hardware, giving the developer the opportunity to realize a healthy profit for his extra development efforts. Just as important is the fact that good assembly language runs perfectly well on slower computers, making the market for such software considerably larger than the market for average software. If you make your software snappy on an 8088, your potential market doubles instantly and the competition thins.

Finally, it's on the slower computers — the PC and AT — that assembly language optimization has the most effect (see the example later in this article), and that's precisely where improved performance is most needed.

Enter the User

So assembly language produces the best code. What of it? If high-level languages make it easier and faster to create programs, who cares if those programs are slower?

The user, that's who. Users care about perceived performance — how well a program seems to run. Perceived performance includes lack of bugs, ease of use, and, right at the top of the list, responsiveness. Hand users a whiz-bang program that makes them wait at frequent intervals, and they'll leave it on the shelf after trying it once. Give users a program that never gets in their way, and they may love it without ever knowing quite why. In these days of all-too-sluggish graphical interfaces, the performance issue is central to the usability of almost every program.

What users don't care about is how a program was made. Do you care how your car was designed? You care that it's safe, that it's reliable, and that it performs adequately, but you certainly don't care whether the manufacturer used just-in-time manufacturing, or whether mainframe or microcomputer CAD was used in the design process. Likewise, users don't care whether a programmer used OOP or C or Pascal, or COBOL, for that matter; they care that a program does what they need and performs responsively. That's not purely a matter of speed, but without speed the user will never be fully satisfied. And when it comes to speed, assembly language is king.

Use Only as Directed

When you need it, there's no substitute for assembly language, but it can be a drag when you don't need it — so know when to use it. Humans are better large-scale designers and small-scale optimizers than compilers, but they're not very good at the grunt work of compiling, such as setting up stack frames, handling 32-bit values, allocating and accessing automatic variables, and the like. Moreover, humans are much slower at generating code, so it's a good idea to avoid being a "human compiler." Some people create complex macros and assembly language programming conventions and do all their programming in assembly language. That works — but what those macros and conventions do is make assembly language function much like a high-level language, so there's no great benefit, especially given that you can drop into assembly language from a high-level language at any time just by calling an assembly language subroutine (or, better yet, by using in-line assembly language in a compiler that offers that feature, such as Turbo C). Unless you're a masochist, let your favorite compiler do what it's best at — compiling — and save assembly language for those small, well-defined portions of your software where your efforts and unique skills pay off handsomely.

A relevant point is that assembly language alone is not the path to performance. If you have a program that takes as long as a second to update the screen, you have problems that assembly language alone won't solve: Proper overall design and algorithm selection are also essential. However, most software designers consider the job done when the design and algorithm phases are complete, leaving the low-level optimization to the compiler. I repeat: No compiler can match a good assembly language programmer at low-level optimization. Given the irregular nature of the 80x86 family and the huge PC software market, it's well worth the time required to hand-optimize the few critical portions that control perceived performance. Only in assembly language can you take full responsibility for the performance of your code.

Don't Spit into the Wind

While I can't offer a cut-and-dried dictum on when to use assembly language, the practice of using it when the user would notice if you didn't is a good rule of thumb. While some programmers would take this rule too far and use assembly language too often, the vast majority of programmers will lean over backwards the other way, in the face of all evidence to the contrary. Hal Hardenberg's late, lamented DTACK Grounded reveled in the folly of the AT&T programmers who implemented the floating-point routines for a super-micro in C rather than assembly language — with the result that the computer performed floating-point arithmetic not quite so fast as a Commodore VIC-20!

Likewise, I once wrote an article in which I measured the performance of an assembly-language line-drawing implementation at four to five times that of an equivalent C implementation. One reader rewrote the C code for greater efficiency, ran it through Microsoft C rather than Turbo C, and wrote to inform me that I had shortchanged C; assembly language was actually "only" 70 percent faster than C. As it happens, the assembly-language code wasn't fully optimized, but that's not the important point: What really matters is that when programmers go out of their way to produce code that's nearly twice as slow (and in an important user-interface component, no less) in order to use a

(continued from page 18)
high-level language rather than assembly language, it's the user who's getting shortchanged. Commercial developers in particular can't afford to ignore this, and I suspect that most such developers are *DDJ* readers. If you're aiming to sell hundreds of thousands of copies of a program, you're guaranteed to have stiff competition. If you don't go the extra mile to provide snappy response, someone else will — and you'll be left out in the cold.

Assembly language lives, though, for one simple reason: Properly applied, it produces the best code of any language. By far

On the other hand, assembly language code is harder and slower to write, and pays off only in the few most critical portions of any program. There are limits to the levels of complexity humans can handle in assembly language, and limits to the development time that can be taken before a product must come to market. Identify the parts of your programs that significantly affect the performance perceived by the user (a code profiler can help greatly here), and focus your efforts on that code, with especially close attention to oft-repeated loops.

80x86 Assembly Language in Action

Enough talk. Let's look at an example of assembly language in action. Listing One, page 94, shows a C subroutine, *CopyUppercase*, that copies the contents of one *far* zero-terminated string to another *far* zero-terminated string, converting all lowercase characters to uppercase in the process. The subroutine consists of a single, extremely compact loop that should be ideal for compiler optimization. In fact, I organized the loop for the best results with Microsoft C 5.0, the test compiler, and used the intermediate variable *UpperSourceTemp* in order to allow for more efficient compiled code. There may be a more efficient way to code this subroutine, but if you're going to go to the trouble of being compiler-specific and knowing compiler code generation that intimately, why not use assembly language, which provides direct control and gives you the freedom to create the best possible code? Microsoft C 5.0 generates the code shown in Figure 1 from the version of *CopyUppercase* in Listing One when maximum optimization is selected with the /Ox switch. It's not bad code, but neither is it great. The *far* pointers are stored in memory and must be loaded each time through the loop, and a considerable amount of work is expended on determining whether each character is uppercase, although the case check is done with a table look-up, which is generally one of the most desirable 80x86 programming techniques. A serious failing is that none of the 80x86 family's best instructions — the string instructions — are used. The upshot is that Listing One runs in the times listed in Figure 2 on various PC-compatible computers. (All times discussed in this article were measured with the Zen timer described in my book *Zen of Assembly Language*, from Scott, Foresman & Company, modified slightly to work with Microsoft C.)

Can we do better in assembly language? Indeed we can, as Listing Two (page 94), which replaces the C version of

(continued from page 20)

CopyUppercase in Listing One with an assembly language version, illustrates. Listing Two simply keeps both *far* pointers in registers and uses string instructions to access both strings; the return for the 21 assembly-language instructions that do that is a performance improvement ranging from two to three-plus times, as shown in Figure 2. If this code happens to be in a performance-sensitive portion of a program, that's quite a return for a little assembly language.

Now, you may well think that the above example is biased in favor of assembly language, what with the *far* pointers, which assembly language tends to handle much better than do compilers. I would disagree: Almost every PC program now takes advantage of the full 640K of memory, and most of that memory must be accessed via *far* pointers, so access to *far* data is a most important issue to PC developers, and the ability of assembly language to handle *far* data just about as fast as near data is a substantial point in favor of assembly language. In fact, this example is representative of a large class of problems developers face, involving data copying, data transformation, data checking, pointers, and segments. Nonetheless, let's see what happens if we alter *CopyUppercase* to use *near* pointers.

Listing Three (page 94) shows Listing One changed to use *near* pointers. Listing Three, which generates the code shown in Figure 3, is indeed much faster than Listing One; it still takes at least half again as long as Listing Two, but it's closing the gap. By contrast, Listing Two wouldn't much benefit from *near* pointers, because it already keeps the pointers in the registers. Does that mean that for *near* data C almost matches assembly language?

Not a chance. We haven't optimized the assembly language implementation yet; Listing Two is just a straight port of Listing One from C to assembly language. Listing Four (page 94) shows Listing Two converted to use *near* point-

```
_CopyUppercase      proc    near
        push        bp
        mov         bp,sp
        sub         sp,0002
Label1:
        les         bx,[bp+08]
        mov         cl,es:[bx]
        inc         word ptr [bp+08]
        mov         ax,cx
        cbw
        mov         bx,ax
        test        byte ptr [bx+0115],02
        je          Label2
        mov         ax,cx
        sub         al,20
        jmp         Label3
Label2:
        mov         ax,cx
Label3:
        les         bx,[bp+04]
        mov         es:[bx],al
        inc         word ptr [bp+04]
        or          cl,cl
        jne         Label1
        mov         [bp-02],cl
        mov         sp,bp
        pop         bp
        ret

_CopyUppercase      proc    near
```

Figure 1: *The code generated for* CopyUppercase *by Microsoft C 5.0 when Listing One is compiled with the /Ox switch (maximum optimization)*

String type/ Language (Listing)	Execution time in microseconds on 8088	80286	80386
Far strings/C (Listing One)	2258 (1.0)	466 (1.0)	140 (1.0)
Far strings/ASM (Listing Two)	662 (3.4)	150 (3.1)	62 (2.3)
Near strings/C (Listing Three)	1183 (1.9)	282 (1.7)	95 (1.5)
Near strings/ ASM (Listing Four)	574(3.9)	115 (4.1)	50 (2.8)
Near strings/ optimized ASM (Listing Five)	410 (5.5)	85 (5.5)	46 (3.0)

Figure 2: *The execution times of the various C and assembly language implementations of* CopyUppercase *shown in Listings One through Five. For a given listing running on a given processor, the number in parentheses represents the performance of that listing relative to the performance of Listing One on that processor; the higher the value, the better the performance. 8088 timings were performed on an IBM XT; 80286 timings were performed on a 10-MHz one-wait-state AT clone; and 80386 timings were performed on a 20-MHz zero-wait-state 32K-cache Toshiba T5200*

```
_CopyUppercase      proc    near
        push        bp
        mov         bp,sp
        sub         sp,0002
        push        di
        push        si
        mov         di,[bp+04]
        mov         si,[bp+06]
Label1:
        mov         cl,[si]
        inc         si
        mov         ax,cx
        cbw
        mov         bx,ax
        test        byte ptr [bx+0115],02
        je          Label2
        mov         ax,cx
        sub         a1,20
        jmp         Label3
        nop
Label2:
        mov         ax,cx
Label3:
        mov         [di],al
        inc         di
        or          cl,cl
        jne         Label4
        mov         [bp+04],di
        mov         [bp+06],si
        mov         [bp-02],cl
        pop         si
        pop         di
        mov         sp,bp
        pop         bp
        ret
_CopyUppercase      proc    near
```

Figure 3: *The code generated for* CopyUppercase *by Microsoft C 5.0 when Listing Three is compiled with the /Ox switch (maximum optimization)*

(continued from page 22)
ers, plus a couple of twists. First, two bytes are loaded, converted to uppercase, and stored at once, cutting the number of memory-accessing instructions in half. Second, the value used to convert characters to uppercase and the upper- and lowercase bounds are stored in registers outside the loop, so that they can be used more efficiently inside the loop. These are simple optimizations, but ones that I doubt you'll find a compiler using — and they're highly effective. As Figure 2 indicates, Listing Four is approximately 20 percent faster than Listing Two and about two times faster than the *near* C implementation of Listing Three.

If you have a program that takes as long as a second to update the screen, you have problems that assembly language alone won't solve

We're not done optimizing yet, though. We've focused so far on relatively simple, linear optimization. Let's pull out all the stops, throw some unorthodox techniques at the problem, and see what comes of it.

On most PC compatibles, the key is this: The processor is slow at fetching instruction bytes and branching (in fact, all 80x86 processors are relatively slow at branching). If we can keep one or the other of those aspects from dragging the processor down, we can often improve performance considerably. As it happens, we can attack both bottlenecks. Look-up tables shrink code size, thereby easing the instruction fetching problem, and avoid branches as well. Well then, why not simply look up the uppercase version of each character? While we're at it, why not look it up with the remarkably compact and efficient *xlat* instruction? In this way we can convert the five instructions used to convert to uppercase in Listing Four to a single *xlat*. We can also improve performance by repeating multiple instances of the contents of the loop in-line, one after the other; doing this allows virtually all of the conditional jumps to fall through, eliminating branching almost entirely. Both changes appear in Listing Five, page 94. As Figure 2 indicates, those two changes improve performance by 8 to 40 percent — and the improvement is greatest on the slower 8088 and 80286 machines, which is surely where speed matters most. (Nor is this code maxed out even yet; I simply had to draw the line somewhere in the interests of keeping the code readily comprehensible and this article to a reasonable length. For example, we could use *lodsw* to speed up Listing Five much as we did in Listing Four. Never assume that your code is fully optimized!)

Bear in mind, too, that the code in Listing Five can handle *far* pointers as easily as *near* if the look-up table is moved into the code or stack segment and accessed with a segment override, a change that would scarcely affect performance at all. When it comes to handling *far* strings, then, we've improved performance by three to five and one-half times. To put that in perspective, the performance improvement gained by running the original C code on a 20-MHz zero-wait-state 32K-cache 80386 computer rather than a run-of-the-mill 10-MHz one-wait-state 80286 computer was only a little over three times. I think it's obvious which is the cheaper solution to improving performance.

(It's worth noting that carefully crafted assembly language

(continued from page 24)
was required to produce the massive performance improvement measured earlier. Assembly language by itself guarantees nothing, and bad assembly language, which is easy to write, brings new meaning to the word bad.)

Don't think I've picked an example that stacks the deck in favor of assembly language. In fact, assembly language would do considerably better if we worked with arrays or fixed-length Pascal-style strings, and would do better than compiled code in cases where there were more variables to keep in the registers. We also weren't able to use repeated string instructions in the earlier example; when such instructions can be used, as is often the case when an entire program's data structures are organized with efficient assembly language code in mind, the performance advantage of assembly language can approach an order of magnitude. In short, we looked at a simple, limited example (and actually one that lends itself relatively well to compiler optimization), and in optimizing it we've scarcely begun to tap the treasure trove of assembly-language tools and techniques.

Yes, compiler library functions can use string instructions and other assembly-language tricks as readily as your own assembly language code can, but there's a great deal that library functions can't do. Don't assume that library functions are well written, either — some are, but many aren't. And remember that the author of the library knows no more than the author of the compiler about when you most need performance, and so must design code for adequate performance under all circumstances. You, on the other hand, can precision-craft your code for best performance exactly when and where you need it. Also, keep in mind that library functions can work only within the current model. When you're working with data on the *far* heap in a program compiled with the small model (an efficient arrangement for programs that must handle a great deal of data), library functions can't help you.

Finally, Microsoft C is a very good optimizing compiler, considerably better than most of the compilers out there. There are a few compilers that generate somewhat better code than Microsoft C, but I'm willing to bet that most of the C programmers reading this use either Microsoft or Turbo C. (Turbo C did not match Microsoft C on this particular example, so I used Microsoft C in order to give C every advantage.) The C code was written to allow for maximum optimization (the loop is only four lines long, for goodness' sake) and uses a macro — not a function call — that expands to a table look up. In other words, the cream of the C crop, given readily optimized code and using a look-up table, went head-to-head with a few dozen hand-optimized assembly-language lines — and proved to be about two to five times slower.

Size Matters Too

I've focused on performance so far because the primary use of assembly language lies in making software faster. Assembly language can make for far more compact programs as well, although that's less often important because the PC has a large amount of memory available relative to processing power and because saving space is a diffuse effort, requiring attention throughout the program, while enhancing performance is a localized phenomenon, and so offers a better return on programming time.

There are cases where program size is crucial — memory-resident programs, device drivers, utilities, for example — and assembly language can work wonders. Of course, good assembly language code is very tight, and hence very small, but there's more to it than that. It's easy to drive programs with compact data strings in assembly language (see "Roll your Own Minilanguages with Mini-Interpreters" which I co-authored with Dan Illowsky, *DDJ*, September 1989). It's also easy to map in code sections from disk as needed; assembly language can be far more flexible than any overlay manager. Finally, assembly language eliminates the need for non-essential start-up and library code. Co-workers tell me of the time they needed to distribute a program to accept a keypress from the user and return a corresponding error level to a batch file. Written in C, the program was 8K in size; unfortunately, the distribution disk didn't have that much free space. Rewritten in assembly language, the same program was a mere 50 bytes long.

When you absolutely, positively need to keep program size to a minimum, assembly language is the way to go.

Can Live with It, Can't Live without It

Assembly language isn't the be-all and end-all of PC programming, but it is the only game in town when either performance or program size is paramount. Assembly language should be used only when needed and, used wisely, offers unparalleled code quality and an excellent return for programming time invested.

For all the drawbacks of assembly language, eight-plus years of PC software development have proven that developers can live with it; programs containing assembly language have been written in an expeditious manner and work very well, indeed. Those same years have shown that developers can't afford to live without assembly language. I suspect you'd be hard pressed to find any important PC software that contains no assembly language at all, and I can assure you that any application with a graphical user interface either contains assembly language or is a dog. (Sure, Windows applications and applications that link in third-party libraries may not contain assembly language, but that's because they've passed that responsibility off to other developers. And just who are those developers? *DDJ* readers, that's who. Somebody has to create the good code that top-notch software requires.)

For all the wishing, 80x86 assembly language isn't going away soon; in fact, it's not going to go away at all. The 80x86 architecture lends itself beautifully to assembly language, and performance will always be at a premium, no matter how fast processors get. Back, when I used a PC, I thought if I had a computer that was ten times faster, all my software would run so fast that I'd never have to wait. Well, now I use just such a computer, and much of the software I use is faster as well (MASM, for example, is about ten times faster than it used to be, and TASM is even faster) — and still I spend a lot of time waiting. Software is never fast enough, and better software is one heck of a lot cheaper than better hardware.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 94.)

Vote for your favorite feature/article.
Circle Reader Service **No. 1.**

Assembly Language
Tricks of the Trade

Hand-picked code for smaller, faster programs

Tim Paterson

It is the nature of assembly language programmers to always look for ways to make their programs faster and smaller. Over the years, the individual programmer develops a personal catalog of tricks and techniques that squeeze out a few bytes here or a few clocks there. My own catalog of 8086 tricks has been 13 years in the making, including a few from the 8080 that survived the translation.

One of the original motivations for finding some of these alternatives to the obvious approach is the severe "branch penalty" of the 8086 and 8088. When a conditional jump is taken on the 8086/8088, four times as many clock cycles are required (16) as when the jump is not taken. However, this penalty has been reduced on the 286 and 386. When taking a conditional jump, the newer processors require only seven clocks, plus one clock for each byte in the instruction at the target of the jump. That is, if you're jumping to an instruction that is 2 bytes long, the conditional jump takes nine clocks. This improvement means that several of the nine tricks presented here are of little or no value on the 286 and 386. However, I have presented them anyway so you'll know what they do if you see them. They are also still useful for code targeted to the 8086/8088.

For each of these tricks, I have compared its size and speed to the "direct" approach. Because the 286 is now the largest selling processor in PCs, I have used 286 clock counts to compare timing. When conditional jumps branch out of the presented code sequence, I assume the target instruction is 2-bytes long so that the branch would take nine clocks.

Tim is the original author of MS-DOS, Versions 1.x, which he wrote in 1980–82 while employed by Seattle Computer Products and Microsoft. He was also the founder of Falcon Technology, which was eventually sold, to Phoenix Technologies, the ROM BIOS maker. He can be reached through the DDJ *office.*

#1 Binary-to-ASCII Conversion

Converts a binary number in *AL*, range *0* to *0FH*, to the appropriate ASCII character.

```
  add   al,"0"                ;Handle 0 - 9
  cmp   al,"9"                ;Did it work?
  jbe   HaveAscii
  add   al,"A"-("9"+1)        ;Apply correction for 0AH-0FH
HaveAscii:
```

Direct approach: 8 bytes, 12 clocks for *0AH-0FH*, 15 clocks for *0–9*.

```
  add   al,90H    ;90H - 9FH
  daa             ;90H - 99H, 00H-05H+CY
  adc   al,40H    ;0D0H - 0D9H+CY, 41H - 46H
  daa             ;30H - 39H, 41H-46H = "0"-"9", "A"-"F"
```

Trick: 6 bytes, 12 clocks.

#2 Absolute Value

Find absolute value of signed integer in AX.

```
  or    ax,ax         ;Set flags
  jns   AxPositive    ;Already the right answer if positive
  neg   ax            ;It was negative, so flip sign
AxPositive:
```

Direct approach: 6 bytes, 7 clocks if negative, 11 clocks if positive.

```
  cwd                 ;Extend sign through dx
  xor   ax,dx         ;Complement ax if negative
  sub   ax,dx         ;Increment ax if it was negative
```

Trick: 5 bytes, 6 clocks.

(continued on page 32)

(continued from page 38)

#3 Smaller of Two Values ("MIN")

Given signed integers in AX and BX, return smaller in AX.

```
  cmp   ax,bx
  jl    AxSmaller
  xchg  ax,bx         ;Swap smaller into ax
AxSmaller:
```

Direct approach: 5 bytes, 8 clocks if *ax* >= *bx*, 11 clocks otherwise.

```
  sub   ax,bx     ;Could overflow if signs are different!!
  cwd             ;dx = 0 if ax >= bx, dx = 0FFFFH if ax < bx
  and   ax,dx     ;ax = 0 if ax >= bx, ax = ax - bx if ax < bx
  add   ax,bx     ;ax = bx if ax >=bx, ax = ax if ax < bx
```

Trick: 7 bytes, 8 clocks. Doesn't work if |*ax* - *bx*| > 32K. Not recommended.

#4 Convert to Uppercase

Convert ASCII character in AL to uppercase if it's lowercase, otherwise leave unchanged.

```
    cmp   al,"a"
    jb    CaseOk
    cmp   al,"z"
    ja    CaseOk
    sub   al,"a" - "A" ;In range "a" - "z", apply correction
CaseOk:
```

Direct approach: 10 bytes, 12 clocks if less than "a" (number, capital letter, control character, most symbols), 15 clocks if lowercase, 18 clocks if greater than "z" (a few symbols and graphics characters).

```
  sub   al,"a"          ;Lowercase now 0 - 25
  cmp   al,"z" - "a" +1 ;Set CY flag if lowercase
  sbb   ah,ah           ;ah = 0FFH if lowercase, else 0
  and   ah,"a" - "A"    ;ah = correction or zero
  sub   al,ah           ;Apply correction, lower to upper
  add   al,"a"          ;Restore base
```

Trick: 13 bytes, 16 clocks. Although occasionally faster, it is bigger and slower on the average. Not recommended. Used by Microsoft C 5.1 *stricmp( )* routine.

#5 Fast String Move

Assume setup for a standard string move, with *DS:SI* pointing to source, *ES:DI* pointing to destination, and byte count in CX. Double the speed by moving words, accounting for a possible odd byte.

```
        shr     cx,1      ;Convert to word count
  rep   movsw             ;Move words
        jnc     AllMoved  ;CY clear if no odd byte
        movsb             ;Copy that last odd byte
AllMoved:
```

Direct: 7 bytes, 10 clocks if odd, 11 clocks if even (plus time for repeated move).

```
        shr     cx,1      ;Convert to word count
  rep   movsw             ;Move words
        adc     cx,cx     ;Move carry back into cx
  rep   movsb             ;Move one more if odd count
```

Trick: 8 bytes, 9 clocks if even, 13 clocks if odd (plus time for repeated move). Not recommended.

#6 Binary/Decimal Conversion

The 8086 instruction *AAM* (ASCII adjust for multiplication) is actually a binary-to-decimal conversion instruction. Given a binary number in *AL* less than 100, *AAM* will convert it directly to unpacked BCD digits in *AL* and *AH* (ones in *AL*, tens in *AH*). If the value in *AL* isn't necessarily less than 100, then *AAM* can be applied twice to return three *BCD* digits. For example:

```
  aam               ;al = ones, ah = tens & hundreds
  mov   cl,al       ;Save ones in cl
  mov   al,ah       ;Set up to do it again
  aam               ;ah = hundreds, al = tens, cl = ones
```

AAM is really a divide-by-ten instruction, returning the quotient in *AH* and the remainder in *AL*. It takes 16 clocks, which are actually two clocks more than a byte *DIV*. However, you easily save those two clocks and more with reduced setup. There's no need to extend the dividend to 16 bits, nor to move the value 10 into a register.

The inverse of the *AAM* instruction is *AAD* (ASCII adjust for division). It multiplies *AH* by 10 and adds it to *AL*, then zeros *AH*. Given two unpacked *BCD* digits (tens in *AH* and ones in *AL*), *AAD* will convert them directly into a binary number. Of course, given only two digits, the resulting binary number will be less than 100. But *AAD* can be used twice to convert three unpacked *BCD* digits, provided the result is less than 256. For example:

```
;ah = hundreds, al = tens, cl = ones
    aad             ;Combine hundreds and tens
    mov   ah,al
    mov   al,cl     ;Move ones to al
    aad             ;Binary result in ax, mod 256
```

AAD takes 14 clocks, which is one clock more than a byte MUL. Again, that time can be saved because of reduced setup.

#7 Multiple Bit Testing

Test for all four combinations of 2 bits of a flag byte in memory.

```
    mov     al,[Flag]
    test    al,Bit1
    jnz     Bit1Set
    test    al,Bit2
    jz      BothZero
Bit2Only:
    . . .

Bit1Set:
    test    al,Bit2
    jnz     BothOne
Bit1Only:
```

Direct approach: 15 bytes, up to 29 clocks (to *BothOne*).

The parity flag is often thought of as a holdover from earlier days, useful only for error detection in communications. However, it does have a useful application to cases such as this bit testing. Recall that the parity flag is EVEN if there are an even number of "one" bits in the byte being tested, and ODD otherwise. When testing only 2 bits, the parity flag will tell you if they are equal — it is EVEN for no "one" bits or for 2 "one" bits, ODD for 1 "one" bit.

The sign flag is also handy for bit testing, because it directly gives you the value of bit 7 in the byte. The obvious drawback is you only get to use it on 1 bit.

(continued on page 34)

(continued from page 32)

```
        test        [Flag],Bit1 + Bit2
        jz          BothZero
        jpe         BothOne ;Bits are equal, but not both zero
;One (and only one) bit is set
.erre  Bit1 EQ 80H          ;Verify Bit1 is the sign bit
        js          Bit1Only
Bit2Only:
```

Trick: 11 bytes, up to 21 clocks (to *Bit1Only*).

Note that the parity flag is only set on the low 8 bits of a 16-bit (or 32-bit 386) operation. Suppose you test 2 bits in a 16-bit word, where 1 bit is in the low byte while the other is in the high byte. The parity flag will be set on the value of the 1 bit in the low byte — EVEN if zero, ODD if one. This is potentially useful in certain cases of bit testing, as long as you are aware of it!

Another example of using dedicated bit positions is to assign flags to bits 6 and 7 of a byte. Then test it by shifting it left 1 bit. The carry and sign flags will directly hold the values in those 2 bits. In addition, the overflow flag will be set if the bits are different (because the sign has changed).

Finally, there is a way to test up to 4 bits at once. Loading the flag byte into *AH* and executing the *SAHF* instruction will copy bits 0, 2, 6, and 7 directly into the carry, parity, zero, and sign flags, respectively.

#8 Function Dispatcher

Given a function number in a register with value *0* to *n – 1*, dispatch to the respective one of *n* functions.

```
;Function number in cx
    jcxz        Function0
    dec         cx
    jz          Function1
    dec         cx
    jz          Function2
    . . .
```

Direct approach 1: *3*n* - 4 bytes, *5*n* clocks maximum. Not bad for small *n (n < 10)*.

```
;Function number in bx
    shl         bx,1
    jmp         tDispatch[bx]
```

Direct approach 2: *2*n* + 6 bytes, 15 clocks. The best approach for large *n* when speed is a consideration.

```
;Function number in cx
    jcxz    Function0
    loop    NotFunc1
Function1:
    . . .
NotFunc1:
    loop    NotFunc2
Function2:
    . . .
NotFunc2:
    loop    NotFunc3
Function3:
    . . .
```

Trick: *2*n* - 2 bytes, *10*n* - 16 clocks maximum. Slow, but compact.

#9 Skipping Instructions

Sometimes a routine will have two or more entry points, but the only difference between the entry points is the first instruction. For example, the instruction that differs from one entry point to the next could be initializing a register to different values to be used as a flag later on in the routine.

(continued on page 36)

(continued from page 34)

```
Entry1:
        mov     al,0
        jmp     Body

Entry2:
        mov     al,1
        jmp     Body

Entry3:
        mov     al,-1
Body:
```

Direct approach: 10 bytes, 11 clocks (from *Entry1*).

Instead of using jump instructions to skip over the alternative entry points, a somewhat sleazy trick allows you to simply skip over those instructions. The technique goes back at least to 1975 with the first Microsoft Basic for the 8080. It became known as a "LXI trick" (pronounced "liksee"), after the 8080 mnemonic for a 16-bit *move-immediate* into register. Essentially, it allows you to skip a 2-byte instruction by hiding it as immediate data. A variation, the "MVI trick" (pronounced "movie"), uses an 8-bit immediate instruction to hide a 1-byte instruction.

Applied to the 8086, there is another variation. The skip can use a *move-immediate* instruction and destroy the contents of one register, or it can use a *compare-immediate* instruction and destroy the flags. Using the latter case the example above could be code such as this:

```
SKIP2F  MACRO
        db      3DH     ;Opcode byte for CMP AX,<immed>
        ENDM
Entry1:
        mov     al,0
        SKIP2F          ;Next 2 bytes are immediate data
Entry2:
        mov     al,1
        SKIP2F          ;Next 2 bytes are immediate data
Entry3:
        mov     al,-1
Body:
```

The effect of this when entered at *Entry1* is:

```
Entry1:
      mov           al,0
      cmp           ax,01B0H    ;Data is MOV AL,1
      cmp           ax,0FFB0H   ;Data is MOV AL,-1
Body:
```

Trick: 8 bytes, 8 clocks (from *Entry1*).

This trick should always be hidden in a macro. Here is a more complete macro that requires an argument specifying what register or flags to destroy. The argument is any 16-bit general register or "F" for flags.

```
SKIP2   MACRO   ModReg
IFIDNI  <ModReg>,<f>        ;Modify flags?
        db      3DH         ;Opcode byte for CMP AX,<immed>
ELSE
?_i     =       0
        IRP     Reg,<ax,cx,dx,bx,sp,bp,si,di>
IFIDN   <ModReg>,<Reg>      ;Find the register in list yet?
        db      0B8H + ?_i
        EXITM
ELSE
?_i     =       ?_i + 1
ENDIF   ;IF ModReg = Reg
        ENDM    ;IRP
.errnz  ?_i EQ 8            ;Flag an error if no match
ENDIF   ;IF ModReg = F
        ENDM    ;SKIP2

;Examples
        SKIP2   f           ;Modify flags only
        SKIP2   ax          ;Destroy ax, flags preserved
```

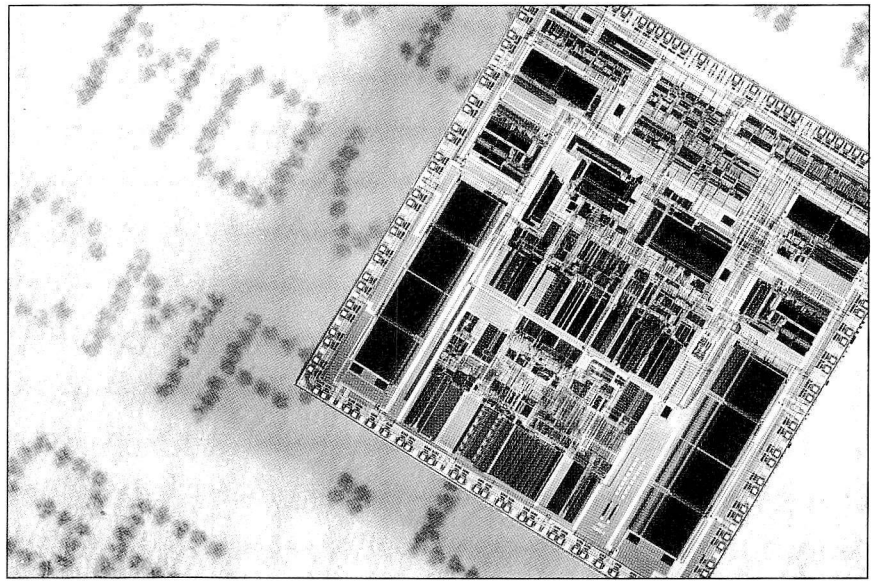
DDJ
Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

68040 Programming

More than just an 030 with floating point

Stephen Satchell

The newest entry in the CPU chip wars is now ready for the system builders: The Motorola 68040. The first available chips will work at 25 MHz, with 33 MHz and faster parts becoming available later this year. Don't think, though, this is just a faster 68030: Motorola built in some nifty features to make multiprocessing hardware much easier to design and build.



68000 Family Overview

Motorola has gone to great pains to make a line of compatible 32-bit microcomputer chips. Like IBM did with the System/360 mainframe computers of the mid-1960s, Motorola made sure that applications code written for the earlier members of the 68000 family would run without modification on later chips. This scheme makes the assumption that programmers segregate I/O and chip control code from the rest of the system.

The general programming model for the 68000 family is the same: Eight 32-bit data registers, seven 32-bit address registers, one 32-bit user stack pointer, one 32-bit supervisor stack pointer, and chip-specific registers. The 68000 family supports operations on individual bits, 8-bit bytes, 16-bit words, 32-bit longwords, and packed binary coded decimal (BCD) data. Address calculations are all 32 bits, although some CPUs have limited addressing capability.

The 68008 (1980) is much the same as the Intel 8088 in that it talks to the outside world over a 20-bit address bus and an 8-bit data bus.

The 68000 (1979), the first CPU in the family, and the low-power CMOS 68HC000 use a 24-bit address bus and 16-bit data bus.

The 68010 (1982) takes the 68000 and adds virtual memory support, using an external memory management unit (MMU) and a special three-instruction "loop mode" that lets the 68010 execute a tight three-instruction loop repeatedly without fetching the instructions from memory more than once.

The 68020 (1984) is the first true 32-bit member of the 68000 family. The address and data busses are both a full 32-bits wide, allowing the chip to directly access four gigabytes (4096 Mbytes) of memory, up to 32 bits at a time. Memory management is provided by an external MMU. Instead of the 68010's "loop mode," the 68020 implements a 256-byte (64 × 4 direct mapped) instruction cache so that most loops run out of on-chip cache memory — improving execution time 33 percent and reducing the load on the system bus. Bit-field instructions let you deal with data of varying bit lengths. Instructions for multiprocessing were added into the 68020 as well.

The 68030 (1987) moves demand-page memory management on-chip, and adds a 256-byte (64 × 4 direct

Steve is free-lance writer and co-founder of Project Notify, a non-profit, emergency communications network. He can be reached at P.O. Box 8656, Incline Village, NV 89450 or on CompuServe at 70007,3351.

(continued from page 38)
mapped) data cache on-chip to complement the 68020's 256-byte instruction cache. The data cache uses a write-through philosophy. The bus system implements a burst transfer mode, that lets the chip effectively use page-mode, nibble-mode, and static-column DRAM to load data and instructions into cache memory quickly.

Enter the 68040

The newest member of the 68000 family, the 68040, essentially combines a beefed-up 68030 and the low-level functions of the 68881 floating-point coprocessor onto the same chip. The improvements, however, go much beyond that. Motorola's goal appears to be to make the 68040 as suitable as possible for large-scale multiprocessing systems.

Instead of one MMU trying to serve the entire chip, the 68040 gives you two: One for instructions, one for data. This keeps data and instruction accesses from causing page table entry faults (not to be confused with page faults) so as to minimize the amount of time the 68040 has to go to RAM to fetch address translation information.

The two on-chip memory caches are completely changed. Not only do you have a 4-Kbyte data cache and a 4-Kbyte instruction cache, but the cache system — particularly the data cache — is designed to minimize the number of times you have to go to the system bus. The two caches are organized as 64 four-way associative maps (256 locations), with 16 bytes of data in each cache location. The data cache can be write through, as it is in the 68030, or the 68040 can use a copyback philosophy that delays the *write* to memory until the chip needs the cache location for something else or the CPU's supervisor empties the cache.

When using cache in a multiprocessing system, you can have data that is one value in cache and another value in main memory. This problem is called "cache coherency." The 68040 takes care of this problem with "bus snooping" — the chip looks at the system bus, and when a *write* memory cycle is detected, any on-chip cache location containing data for the changed location is marked invalid.

What happens, though, when one 68040 has changed data, but hasn't written it back to DRAM yet? The bus snoop hardware has another trick up its sleeve. When a *read* memory cycle is detected, the 68040 checks its data cache to see if it changed the requested location; if so, it inhibits the RAM memory cycle and sends the correct data to the other CPU. This reduces the amount of work programmers have to do to keep data up-to-date.

If you do a lot of scientific work, watch out for the floating-point unit. On the 68040, the only floating-point operations supported are absolute value, add, branch on condition, compare, decrement and branch conditionally, divide, move, move multiple, multiply, negate, nop, restore internal state, save internal state, set on condition, square root, subtract, trap on condition, and test. Other operations supported by the 68881, such as the trig and logarithmic functions, have to be handled by software emulation.

Assembler Programming Considerations

Portability When writing code that needs to run on different systems, you need to limit yourself to those instructions common to all the 68000 family. (See Table 1 for those instructions to avoid.) In particular, pay attention to addressing modes. The 68020, '30, and '40 support some additional modes not found on the '00, '08, and '10. Also try to segregate chip-dependent functions from the rest of your program. This limits how much code has to be replaced as you shift from CPU to CPU. The majority of your code should be

(continued from page 40)
running in user mode anyway.

Loops The loop mode of the '10 is of limited use, being composed of a loop-able instruction and a DBcc instruction. Use this construct when you can on the off chance you end up running on a '10, such as one of the older Sun workstations. Where possible, try to keep loops under 256 bytes, the size

68010 from 68000 and 68008	
Move from CCR	Move from Condition Code register
Move from SR	Move from Status register
MOVEC	Move Control register
MOVES	Move Status register
RTD	Return and Deallocate
68020 from 68010 Data alignment restriction dropped	
Bcc	Branch conditionally (allow 32-bit displacements)
BFCHG	Test Bit Field and Change
BFCLR	Test Bit Field and Clear
BFEXTS	Bit Field Extract Signed
BFEXTU	Bit Field Extract Unsigned
BFFFO	Bit Field Find First One-bit
BFINS	Bit Field Insert
BFSET	Test Bit Field and Set
BFTST	Test Bit Field
BKPT	Breakpoint
CALLM	Call Module
CAS	Compare and Swap Operands
CAS2	Compare and Swap Dual Operands
CHK2	Check register against upper and lower bound
CMP2	Compare register against upper and lower bound (between)
cpBcc	Branch on CoProcessor condition
cpDBcc	Test CoProcessor condition Decrement and Branch
cpGEN	CoProcessor General function
cpRESTORE	CoProcessor Restore functon
cpSAVE	CoProcessor Save function
cpScc	Set on CoProcessor condition
cpTRAPcc	Trap on CoProcessor condition
DIVSL	Long signed divide
DIVUL	Long unsigned divide
EXTB	Extend byte to long
PACK	Pack binary coded decimal (BCD)
RTM	Return from Module (*not* "Read the manual")
TRAPcc	Trap conditionally
UNPK	Unpack binary coded decimal (BCD)
68030 from 68020	
(CALLM)	
PFLUSH	Invalidates specific entry in the address translation cache (ATC)
PFLUSHA	Invalidates all entries in the address translation cache (ATC)
PLOAD	Load an entry into the address translation cache
PMOVE	Load an entry into the address translation cache
PTEST	Get information about a logical address
(RTM)	
68040 from 68030	
CINV	Invalidate cache entries
(cpBcc)	
(cpDBcc)	
(cpGEN)	
(cpRESTORE)	
(cpSAVE)	
(cpScc)	
(cpTRAPcc)	
CPUSH	Push, then invalidate, cache entries
Floating-point instructions	
MOVE16	Move 16-byte block; block must be aligned
(PFLUSHA)	
(PLOAD)	
(PMOVE)	

Table 1: *680x0 family instruction set differences. An instruction or capability added or changed is in the open. An instruction or capability removed is in parens. For example, the CALLM instruction was removed in the 68030, so in the table it shows as (CALLM).*

(continued from page 42)
of the instruction cache on the '20. If a much-repeating loop can't be squeezed down that far, move seldom-executed code such as exception code outside of the loop. The longer you can stay in the cache, the faster that loop executes.

Loop Data In assembler, it is usually easier to whip through an array word by adjacent word, so most assembler language programmers won't have to concentrate on what order data gets accessed. If you are writing a table-driven package, though, pay attention to how table information makes you access data. Where possible, the table should be optimized so your program sweeps through any array. This is somewhat important on the '30, and much more important on the '40 — particularly in multiprocessing systems.

Tests Many times, you have to load one of two values into a register or location based on some test condition. The "IF . . . THEN . . . ELSE . . . " construction is easy to understand, but the multiple branches can play hob with instruction fetching. Instead, try " . . . IF . . . THEN . . . " where you set the less common value, perform the test, and conditionally branch around the more common value. The penalty on '00, '08, and '10 CPUs is almost zero, but the savings on the '20, '30, and '40 can be significant. In fact, the first way requires at least five instructions (test, branch-false, set-1, branch, set-2) while the other way saves one instruction (set-2, test, branch-false, set-1).

High-Level Language Considerations

Portability Chip-dependent functions usually have to be written in assembler, so make sure the design of the system routines are as generic as possible so you don't have to change applications code when the next gee-whiz feature is introduced in the 68050. You'll need to package separate interface modules for each chip. High-level code should always be run in user mode.

Loops If your compiler can optimize for the loop mode on the '10 or if the library includes routines to perform functions using loop mode, use them. When structuring loops that are executed often consider dropping structured programming practices to pack the loop as tight as possible. The goal is to get the loop within the 256-byte window of the instruction cache of the '20. Branches are much cheaper than function calls to get the seldom-used code out of the loop. You have more latitude with the larger cache on the '30 and '40.

Loop Data Be very careful when transversing arrays that you know *exactly* how your compiler is working. Fortran programmers need to remember that they have to vary the first subscript first in order to walk through data sequentially. For PL/I and Pascal programmers, most compilers require you to vary the last subscript first to sweep an array. C programmers need to remember that when accessing a multidimensional array using the array operators that are in the construct "a[i][j]", the fragment "a[i]" loads a pointer, then "<e>[j]" loads the desired word; use an intermediate pointer where possible to limit the amount of pointer loading when the first subscript is held locally constant.

Tests You are at the mercy of the compiler when it comes to ordering tests to save time. Because compilers vary so much in what they do, it probably isn't worth it to change the way you select values.

Conclusion

The 68040 is more than "just a 68030 with floating point" and more than Motorola's weapon to fight the Intel 80486. It is a well-designed product in its own right. Graphics programmers like the support for manipulating bits, particularly the bit-field instructions introduced by the '20 and continued in the '40.

DDJ
Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

Homegrown Debugging—*386 Style!*

Use hardware breakpoints to sniff through your C and assembly code

Al Williams

Although the installed base of 80386-based machines is ever increasing, most use this souped-up machine as a faster 8086. One of the problems in running the 80386 under DOS is that you lose many of the advantages of the 386. In addition, many of the 80386's powerful features are only usable in protected mode. Of course, developers loath to use special 80386 features because this can shut them out of the large 8086/80286 market.

Still, some features are usable while the 80386 is operating as an 8086 (the so-called "real mode"). For instance, the 80386 has powerful on-board hardware that allows sophisticated debugging techniques that require hardware debugging boards on other processors. This on-board hardware is available in real mode (as well as the other modes). With a little ingenuity, you can put this hardware to work while debugging programs.

This article puts a little of that kind of ingenuity in your hands by showing how you can use the 80386 hardware to debug your programs. I'll provide a program that can be included in your assembly code to establish breakpoints for the purpose of debugging either C or assembly language programs. In addition, I'll provide an example program and a quick utility that I'll explain shortly.

Al Williams is a staff systems engineer for Quad-S Consultants Inc. His current work includes a hypertext system, several expert systems, and a 386 DOS extender package. He can be reached at 2525 South Shore Boulevard, Suite 309, League City, TX 77573.

All examples presented in this article compile under either MASM 5.0 or Microsoft C 5.1.

BREAK386

BREAK386 (Listing One, BREAK386.ASM, page 96) is not a traditional debugger in the sense of, say, DEBUG or CodeView. By adding BREAK386 to your assembly language code, you can study it with code, data, and single-step breakpoints. You can also examine DOS or BIOS interrupts that your program calls. In addition, BREAK386 can add the same 386 hardware debugging to your Microsoft C programs.

BREAK386 provides functions to set up 386 debugging (*setup386( )*), set breakpoints (*break386( )*), and reset 80386 debugging (*clear386( )*). In addition, BREAK386 provides an optional interrupt handler (*int1_386( )*) that supports register, stack, and code dumps along with single stepping. You can use any of these functions from either C or assembly language.

There are cases where you may wish to modify *int1_386( )* or write your own interrupt handler. For example, you may want to send the register dumps to a printer and automatically restart your program. With C, you will often want the interrupt handler to print out variables instead of registers. I'll provide some example interrupt handlers in C in a later section.

Using BREAK386

You must assemble BREAK386 before you can use it. Be sure to change the *.MODEL* statement to reflect the model you are using. If you are using explicit segment definitions in assembly, you must decide how to integrate BREAK386's code and data segments with your own. Assemble BREAK386 with the */Ml* option to prevent MASM from converting all labels to uppercase. The resulting .OBJ file can be linked with your programs just as with any other object module.

If you are using programs (such as memory managers or multitaskers) that also use 386-specific functions, you may have to remove these programs before BREAK386 will function. The other program will usually report a "privilege exception" or something similar. Simply remove the other 386 programs and try again.

Adding 386 breakpoints to your program requires three steps:

- Call *setup386( )* to set the debug interrupt handler address
- Set up breakpoints with the *break386( )* call
- Call *clear386( )* before your program returns to DOS

Note that when calling these routines from assembly, the routine names contain leading underscores. For convenience, Listing Two (BREAK386.INC, page 102) contains the assembly language definitions to use BREAK386. Listing Three (BREAK386.H, page 102) contains the same definitions for C. BREAK386.INC also includes two macros, *traceon* and *traceoff*, which are used to turn single stepping on and off from within the program.

Figure 1 shows the output from a breakpoint dump when using *int1_*

(continued from page 46)

386( ). The hexadecimal number on the first line is the contents of the low half of the DR6 register at the time of the breakpoint. The display shows all 16-bit and segment registers (except FS and GS). Following that is a dump of 32 words of memory starting at the bottom of the stack (1CB1:09FA in the example). The first three words of the stack are from the debug interrupt. The first word is the IP register, followed by the CS register and the flags. A simple change in the interrupt handler can remove this extra data from the display (see "Detailed Program Operation" in the next section).

Below the stack dump is a dump of program code. This dump usually consists of 16 bytes; 8 bytes before the current instruction and 8 bytes at the instruction pointer. This is convenient for data breakpoints because they occur after the offending instruction. The dump shows the starting memory address (1B66:0049) followed by the bytes at that address. An asterisk marks the current CS:IP location, followed by the remaining 8 bytes. If IP is less than 8, the code dump will start at CS:0 resulting in fewer than 8 bytes before the asterisk.

The last line of the dump prompts you for further action. You can:

1. View your program's output screen. When you select this option, BREAK386 replaces the current screen with your program's original output. To restore the debugging screen, press any key.

```
Program breakpoint:OFF1
AX=0000 FL=7216 BX=0080 CX=0007 DX=06AA
SI=0000 DI=0A00 SP=09FA BP=0882
CS=1B66 IP=0051 DS=1BAD ES=1B56 SS=1CB1
Stack dump:( 1CB1 : 09FA )
0051 1B66 7216 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

CODE=1B66 : 0049 =6A 04 E8 3F 00 83 C4 08 * B9 14 00 8A D1 80 C2 41

<V>iew output, <T>race toggle, <C>ontinue or <A>bort? _
```

Figure 1: *Sample output from a breakpoint dump*

2. Toggle the trace flag. This will switch the state of the trace or single-step flag, and continue the program in the same manner as the "C" command (see number 3). To determine whether or not tracing is on, examine the value of DR6. If bit 14 is set (4000 hex), tracing is on.
3. Continue execution of the program. Selecting this option will resume the program where it left off. The program will execute until the next breakpoint (if the trace flag is clear) or to the next instruction (if the trace flag is set).
4. Abort the program. This will cause the program to exit. Be careful, however, when using this selection. If you have interrupt vectors intercepted, expanded memory allocated, or anything else that needs fixing before you quit, the "A" command will not take care of these things unless you rewrite the interrupt handler or *clear386( )*. (Also, if your program spawns child processes, and the breakpoint occurred in the child, the abort command will terminate the child and the parent program will continue without breakpoints.)

Listings Four and Five, page 102, show examples of using BREAK386 in assembly and C. The identifiers beginning with *BP_* are defined in BREAK386.H and

(continued from page 48)
BREAK 386.INC.

A few notes on these functions are in order. Your program must call *setup386( )* before any other BREAK386 calls. You should pass it a segment and an offset pointing to the interrupt handler. After calling *setup386( )*, you may use *break386( )* to set and clear breakpoints. Figure 2 shows the parameters *break386( )* requires.

You must keep in mind a few facts about the 80386 when setting breakpoints or tracing. First, 2- and 4-byte data breakpoints must be aligned according to their size. For example, it is incorrect to set a 2-byte breakpoint at location 1000:0015 because that location is on an odd byte. Similarly, a 4-byte breakpoint can monitor address 1000:0010 or 1000:0014 but not address 1000:0013. If you must watch an unaligned data item, you will have to set multiple breakpoints. For example, to monitor 2 bytes at 1000:0015, set a 1-byte breakpoint at 1000:0015 and another at 1000:0016.

Also, keep in mind that a data breakpoint will occur even if you only access a portion of its range. For instance, if

80386 Debugging Features

Most PC developers are familiar with some aspect of chip debug assistance. Even the 8088 has a breakpoint interrupt and a "single-step flag," which allows debuggers to trace code one instruction at a time. The 386 shares these same features with the earlier processors, but adds eight debug registers (two of which Intel reserves). These debug registers control the hardware breakpoint features.

Hardware breakpoints are much more powerful than ordinary breakpoints (such as those in DEBUG) for two reasons. First, hardware breakpoints don't actually modify your program. This means that you can set breakpoints anywhere, even in ROM. Also, a program can't overwrite a breakpoint when it modifies itself or loads an overlay. Second, it is possible to set breakpoints on data. A data breakpoint triggers when your program accesses a certain memory location.

Microsoft's CodeView implements a similar data breakpoint capability, called "tracepoints." To maintain compatibility with non-386 PCs, however, CodeView doesn't use 386 features. As a result, CodeView checks tracepoints after the execution of each instruction. This, of course, is terribly slow. By moving the tracepoints to 386 hardware, execution isn't slowed down at all. Actually, you will usually want to slow down execution just a bit (see the discussion of the exact bit). Even then, the slowdown in execution is imperceptible.

Because there are four debug address registers in the 80386, it is possible to have four active breakpoints at once. Each address register (DR0-DR3) represents a linear address at which a different breakpoint will occur. In protected mode, the concept of a linear address is not straightforward. In real mode, however, a linear address can easily be calculated from a segment/offset pair. Simply multiply the segment value by 10 hex (shift left 4 bits) and add the offset. For example, to set a data breakpoint at B800:0020 (somewhere in the CGA video buffer), you would need a linear address of:

B800 × 10 + 20 = B8020

Once you have loaded the address registers, you must enable the breakpoints you wish to use and tell the processor what type of breakpoints they are. This is done via the debug control register (DR7). DR7 contains bits to enable each breakpoint and to set their type individually (see Figure 4). You will notice that DR7 has global and local enable bits as well as global and local exact bits (explained shortly). The difference between the various global bits and local bits is only important when the 80386 is multitasking in protected mode. For the purpose of this article, they are the same.

The Exact Bits

The exact bits are flags to tell the 80386 to slow down. At first glance, this doesn't seem to be helpful, but a detailed look at the 80386 architecture reveals the purpose of this bit.

The 80386 gains some of its speed by overlapping instruction fetches and data fetches. This is an excellent idea when executing code, but causes problems in debugging data. Without the exact bit set, a data breakpoint will not occur at the instruction that caused the data access! Being somewhat of an inconvenience, Intel included the GE/LE bits. With either (or both) of them set, data breakpoints will occur immediately after the instruction that caused them, although the processor

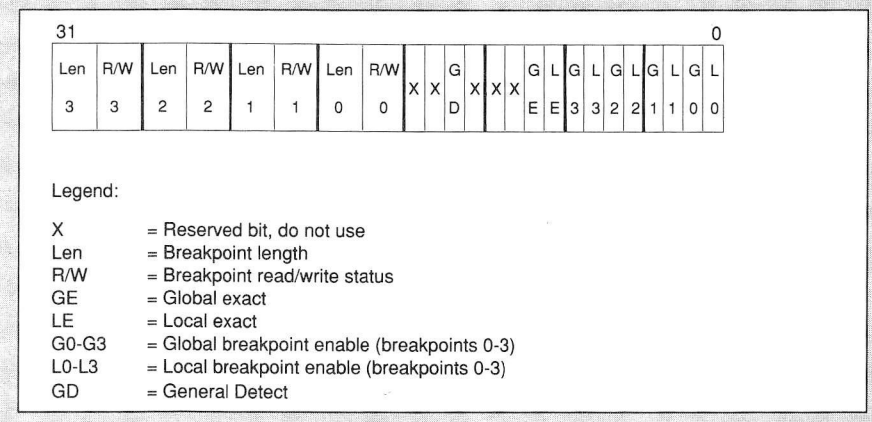


Figure 4: *Bits contained in DR7 to enable and set the type of breakpoints*

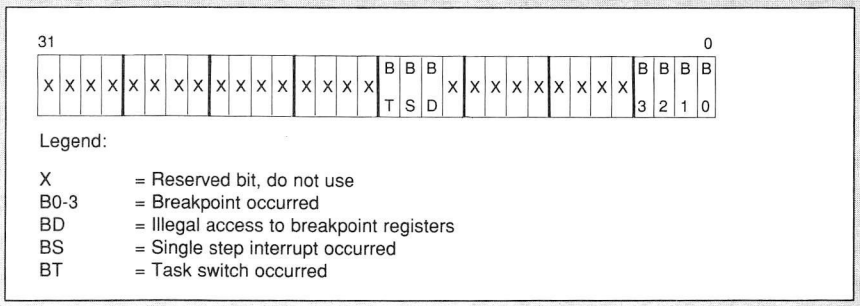


Figure 5: *Bits in DR6 corresponding to the various breakpoints conditions*

The original Volume 15 book had a printing error:
pages 223-254 were missing and
pages 255-286 were repeated twice.

The missing pages were recreated in this PDF by taking the equivalent pages from the March 1990 magazine issue.

you are monitoring a word at 2200:00F0 and a program writes a byte to 2200:00F1, a breakpoint will occur.

Setting a data breakpoint with *break386( )* will also set the global exact bit. When all the data breakpoints are either reassigned or deactivated, *break386( )* will clear the exact bit.

Because *int1_386( )* always sets the resume flag, you will find that a code will lose a slight amount of speed.

Other Bits

All debug breakpoints generate an interrupt 1. To distinguish the various breakpoints, you must read the debug status register (DR6). DR6 has bits corresponding to the various breakpoint conditions (see Figure 5). Note the BT flag at bit 15. As with the local bits in DR7, only multitasking systems use the BT flag. Therefore, the flag is not considered in this article. The 386 never clears the bits in DR6, so after you determine what caused the interrupt, you should clear DR6.

The Only Other Bit We Haven't Discussed is . . .

With the general detect (GD) bit set in DR7, the 80386 prohibits access to the debug registers. Any attempt to access the debug registers will cause an interrupt 1 with the BD flag set in DR6. Intel's in-circuit emulator uses this feature, although you can use it if you have any reason to disable or control access to the debug registers. When a GD interrupt occurs, the interrupt handler is invoked and the GD bit is cleared. Otherwise, the routine would fault (with an endless loop) when the interrupt routine attempted to read DR6.

You can decide from the interrupt routine whether to terminate the user program, or to allow access to the registers. BREAK386 does not use the GD bit.

The Resume Flag

The last consideration with breakpoint interrupts is how to resume the interrupted program. If we simply return (as in a normal interrupt), there is nothing to stop a code breakpoint from occurring again immediately. The resume flag (found in the flag's register) prevents this from occurring. This flag inhibits further debug exceptions while set, and resets automatically as soon as one instruction successfully executes. Control of the resume flag is automatic in protected mode. Handling it from real mode, however, is somewhat of a trick, as seen in BREAK386. — A.W.

breakpoint that immediately follows a data breakpoint won't work. I'll show how this can be rectified shortly.

Because *INT* and *INTO* instructions temporarily clear the trace flag, BREAK386 will not single step through interrupt handlers. If you wish to single step through an interrupt routine, you will have to set a breakpoint on its first instruction. A replacement for *int1_386( )* might emulate *INT* and *INTO* instructions to solve this problem.

Because BREAK386 uses BIOS keyboard and video routines, take care when placing breakpoints in these routines. In addition, single-stepping BIOS keyboard and video routines should be avoided. If you must debug in these areas, reassemble BREAK386 so that it doesn't use BIOS (see the DIRECT equate in BREAK386.ASM). Note, however, that many of its features will no longer function. Finally, you should avoid setting breakpoints in BREAK386's code or data.

BREAK386.INC contains two macros, *traceon* and *traceoff*, that can be used to control tracing. You may insert them anywhere in your code to enable or disable tracing. Remember, however, that you will see the *traceoff* macro as well as your own code when single stepping.

The function *clear386( )* must be called prior to exiting the program. This turns off the breakpoint handlers. If you fail to call *clear386( )* for any reason (a control-break, or a critical error), the next program that uses a location you have breakpointed will cause the break to occur. This can have unfortunate consequences because your interrupt 1 handler is probably no longer in memory. If you find that you have exited a program without turning off debugging and you have not encountered a breakpoint, run DBGOFF (Listing Six, page 104) to turn off hardware debugging.

With some care, BREAK386 can be used with other debuggers. In CodeView, for example, BREAK386 seems to work fine, as long as you are not single stepping (via CodeView). When you single step data breakpoints will be ignored and BREAK386 code breakpoints will "freeze" CodeView at that step. If you are using BREAK386 with CodeView, it is probably a good idea to leave the code breakpoints and single stepping to CodeView.

Detailed Program Operation

BREAK386 (Listing One) begins with the .386P directive, which ensures that MASM 5.0 will generate references to the debug registers. Be careful to place the .MODEL directive before the .386P,

(continued from page 52)
otherwise 32-bit segments will be generated (which doesn't work well with unmodified DOS!).

The parameters you may want to change are near the top of the source file. The equate to DIRECT controls the video mode. If DIRECT is 0, BREAK386 uses BIOS for input and output. If, however, you want to poke around in the keyboard or video routines, you must set DIRECT to 1. This causes BREAK386 to use direct video output for the debug dump. It will share the screen with your program (no video swapping) and breakpoints will simply terminate the program in a similar manner to the "A" command mentioned earlier.

You can change the STKWRD equate to control how many words are dumped from the stack when using *int1_386( )*. Setting STKWRD to zero will completely disable stack dumping. Similarly, if you set INTSTACK to zero, the display will not show the IP/CS/FLAGS at the top of the stack. If you are writing your own interrupt handler and don't need *int1_386( )*, you can assemble with *ENABLE_INT1* set to zero to reduce BREAK386's size.

The operations of *start386( )*, *clear386( )*, and *break386( )* are fairly straightforward. The implementation of *int1_386( )* deserves some comment. It is important to realize that *int1_386( )* only debugs non-386-specific programs because it only saves the 16-bit registers and the 8086 segment registers (*int1_386( )* does not destroy FS and GS). Because *int1_386( )* only runs on a 386, it does use the 32-bit registers. You can easily modify *int1_386( )* to save all the 386 registers, but it requires more space on the interrupted program's stack.

The most difficult aspect of the interrupt handler is managing the resume flag. The code below label c1 converts the three words at the top of the stack into six words so that setting the resume flag is possible. There are three things to remember about the way the resume flag is managed:

2- and 4-byte data breakpoints must be aligned according to their size. For example, it is incorrect to set a 2-byte breakpoint at location 1000:0015 because that location is on an odd byte

1. As mentioned earlier, *int1_386( )* always sets the resume flag. As a consequence, a code breakpoint that occurs immediately after a data breakpoint will not cause an interrupt. This is due to the resume flag being set even though the instruction that generated the data breakpoint has already executed. When the program restarts, the next instruction will execute with the resume flag set. This could be rectified by not setting the resume flag in the interrupt handler when processing data breakpoints.
2. An interrupt handler written entirely in C has no way to manipulate the resume flag properly. Listing Seven, page 104, however, shows two assembly language functions that allow you to write your handler in C. (See the next section for more details on writing C interrupt handlers.)
3. In real mode, hardware interrupt handlers (for example, those in the BIOS) will probably not preserve the resume flag. This means that if your code runs with interrupts enabled, there is some chance that one breakpoint will cause two interrupts. This chance increases greatly if interrupts remain disabled during the interrupt 1 processing. Why is this true? If the 80386 receives a hardware interrupt just before executing an instruction with the resume flag set, it will process that interrupt. When the interrupt returns, the resume flag is clear and the breakpoint occurs again. When interrupts are disabled during breakpoint processing, it is far more likely that an interrupt is pending when the program restarts. If interrupts were enabled while processing the debug interrupt, however, there is little chance of this happening. If it does, simply press "C" (when using *int1_386( )*).

```
retcode=break386(n,type,address);
where:
   n is the breakpoint number (from 1 to 4).
   type is the type of breakpoint. This should be one of the manifest constants defined in
   BREAK386.H (or BREAK386.INC). If you are clearing the breakpoint, the type is not
   meaningful.
   address is the address to set the breakpoint. This must be a far address (that is, one
   with both segment and offset). If you are using small model C, you should cast the
   pointer to be a far type (see the example). To clear a breakpoint, set address to
   0000:0000 (or a far NULL in C).
   retcode is returned by the function. A zero indicates success. A non-zero value means
   that you tried to set a breakpoint less than 1 or greater than 4. Note that the type
   parameter is not checked for validity.

   The types available are:
         BP_CODE        - Code breakpoint
         BP_DATAW1      - One byte data write breakpoint
         BP_DATARW1     - One byte data read/write breakpoint
         BP_DATAW2      - Two byte data write breakpoint
         BP_DATARW2     - Two byte data read/write breakpoint
         BP_DATAW4      - Four byte data write breakpoint
         BP_DATARW4     - Four byte data read/write breakpoint
```

Figure 2: *The parameters required by* break386( )

Advanced Interrupt Handlers in C

It is possible to write an interrupt handler completely in C to monitor data breakpoints. The handler must be declared as a *far* interrupt function. For example, the following function could be linked with the example in Listing Five:

```
void interrupt far
new1(Res,Rds,Rdi,Rsi,Rbp,Rsp,Rbx,Rdx,
                                Rcx,Rax)
   {
   printf("\nBreakpoint reached.\n");
   }
```

By calling *setup386new1( )* instead of *setup386(int1_386)*, *new1( )* will be invoked for every breakpoint. Your function can read and write the interrupted program's registers using the supplied parameters (*Rax*, *Rbx*, and so on). Keep in mind that you cannot use this technique for code breakpoints. C's inability to manipulate the resume flag will cause an endless loop on a code breakpoint.

Listing Seven, provides the functions to write interrupt handlers in C. The procedure is much the same as described earlier, except that you must

(continued on page 57)

(continued from page 54)
call *csetup386( )* instead of *setup386( )*. The argument to *csetup386( )* is always a pointer to an ordinary *far* function (even in small model).

The actual interrupt handler is *_cint1_386( )*. This function will call your C code when an interrupt occurs. *_cint1_386( )* passes your routine two arguments. The first argument, a *far* void pointer, is set to the beginning of the interrupted stack frame (see Figure 3 for the format of the stack frame). The second argument is an unsigned long *int* that contains the contents of DR6.

All registers, and local variables on the stack can be read using the pointer to the stack frame (if you know where to look). In addition, all values (except SS) can be modified. It is usually wise not to modify SP, CS, or IP.

_cint1_386( ) switches to a local stack. The size of the stack can be controlled using STACKSIZE (near the top of Listing Seven). Be sure to adjust the stack if you need more space.

Listing Eight (page 105) shows an example of an interrupt handler in C. The example interrupt handler displays a breakpoint message and allows you to continue with or without breakpoints, abort the program, or change the value of a local variable in the *loop( )* function.

Future Directions

Many enhancements and modifications are possible with BREAK386. By altering the words on *int1_386( )*'s stack, for example, you can modify registers. You can redirect output to the printer (although you can screen print the display now) by replacing the OUCH routine. Perhaps the most ambitious enhancement would be to use BREAK386 as the core of your own debugger. You could write a stand-alone debugger or a TSR debugger that would pop up over another debugger (DEBUG or CodeView).

Keep in mind that 386 hardware breakpoints aren't just for debugging. The data breakpoint capability has many uses. For example, you might want to monitor the BIOS keyboard typeahead buffer's head and tail pointers to see when a keystroke is entered or removed. In this manner you could capture the keyboard interrupt in such a way that other programs couldn't reprogram your interrupt vector.

You can also use data breakpoints to detect interrupt vector changes or interrupt processing. Some assembly language programs could use data breakpoints for automatic stack overflow detection. Programs that decrement the stack pointer without using a push instruction (Microsoft C programs, for example) are not candidates for this type of stack protection.

Debugging with 386 assistance is quite practical and useful. The programs presented here should get you started and help you develop your own programs with this powerful hardware feature.

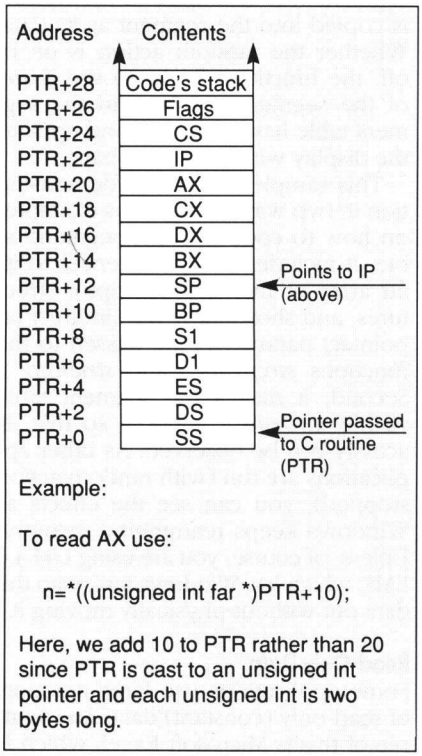


Figure 3: *Stack frame passed to the C interrupt handler*

Bibliography

Turley, James L., *Advanced 80386 Programming Techniques*, Osborne McGraw-Hill, Berkeley, Calif., 1988.

Intel Corporation, *80386 Programmer's Reference Manual*, Intel Corp., Santa Clara, Calif., 1986.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 96.)

Vote for your favorite feature/article.
Circle Reader Service **No. 4.**

Part II

Managing Multiple Data Segments Under *Microsoft Windows*

The segment table provides a little-known way of managing multiple data segments

Tim Paterson and Steve Flenniken

In last month's installment, we presented a method for managing multiple data segments under MS Windows using a little-known Windows feature, the segment table, along with a library of macros and functions to assist in applying the technique. For this month's installment, we've prepared a sample Windows program called "segments" that demonstrates the *segtable* library. In its "random action" phase, it randomly allocates, reallocates, and frees global memory. A window displays statistics about each memory block, including its *pSeg* (the address of its *SegmentTable* entry), the current segment number, the previous segment number, and the number of times it has moved since it was allocated. A timer function is used to keep the window continuously updated, even when another application has the input focus.

The sample application in Listing One (page 106) uses one segment as the place to keep track of all the other segments that it fiddles with and displays in the window. That segment contains an array of structures, one for each additional segment. Because it is referenced so often, the macro *FARDATAP* is defined to return the far pointer to the first structure in this segment. Listings Two through Five (beginning on page 108) provide the rest of the files required by the application.

The menu bar is used to start and stop the random action mode. When on, the timer function picks one of the structures. If the structure does not yet have a *pSeg*, it allocates one with a random amount of memory. If it already has a *pSeg*, it will do one of three things: Reallocate the *pSeg* with a different memory size; free the data, but keep the *pSeg*; or free the *pSeg* altogether. Whenever a segment is allocated or reallocated, a text string containing the last action ("A" for allocate or "R" for reallocate) and the size of the segment (for example, "1484 bytes") is copied into the segment as its data. Whether the random action is on or off, the function checks to see if any of the segment numbers in the segment table have changed, and updates the display window if they have.

This sample is a useful demonstration in two ways. First, it has examples on how to code with the segment table. It includes many references to its far array of memory descriptor structures, and shows how IFP (indirect far pointer) parameters are passed to the functions *strcpyifp( )* and *strlenifp( )*. Second, it makes the segment table visible through a window so that its activity can be observed. As other applications are run (with random action stopped), you can see the effects as Windows keeps rearranging memory. Unless, of course, you are using LIM 4.0 EMS, which lets Windows just swap the data out without physically moving it.

Read-Only Data

Some applications use large amounts of read-only (constant) data. An example of this is Microsoft Excel, which is written in C and compiled into pcode, not native 8086 code. The pcode is

Tim is the original author of MS-DOS, Versions 1.x, which he wrote in 1980-82 while employed by Seattle Computer Products and Microsoft. He was also the founder of Falcon Technology, which was eventually sold to Phoenix Technologies, the ROM BIOS maker. Steve formerly worked at Seattle Computer Products, Rosesoft (makers of ProKey), and is now with Microrim, working with OS/2 and Presentation Manager. Both can be reached c/o DDJ.

(continued from page 58)
data, not code, because it is never actually executed. Other applications could simply have large amounts of data in the form of tables or other structures.

Like code, read-only data should be marked as discardable in the linker definition file. This allows Windows to throw it away to make room, but reload it from disk later when needed. Another good practice is to keep segment size to less than 16K, the size of the LIM 3.2 expanded memory page frame. Windows can then choose to use space in EMS for those segments that fit, entirely transparent to the application.

Code and read-only data don't sound any different so far, but there is an important distinction. Windows keeps track of how often each code segment is used, in order to help it make a good decision on discarding one when it needs to free some memory. It does this with the reload thunk. Every far call to a discardable segment actually calls a thunk specific to that entry point. If the segment being called is present in memory, the thunk will contain a jump to the entry point. If the segment is not loaded, the thunk will cause Windows to load it. Either way, the thunk also notes the fact that a call to that segment was made. Windows uses a least-recently used (LRU) algorithm for determining the best segment to discard when memory is needed. The thunks are the source of its information.

The easiest way to deal with discardable read-only data segments is to put a little code in them. These lines of assembly language belong in each segment (but with a unique entry point name for each):

```
Load_This_Segment:
    mov     ax,cs
    retf
```

To ensure that a segment is loaded, and to find out where, call this entry point. The return value in *ax* is the segment of the data. The call to this entry point is, of course, actually a call to a thunk that ensures the segment is loaded.

The segment number returned by this call can be stuffed into an empty entry in *SegmentTable* so that it will stay updated in case of movement. But recall that this segment can also be discarded. In that case, Windows will update the segment table with the (even-numbered) handle for that segment. This complicates things a bit. Now we could make a reference to the segment table and find an even number, indicating that the segment we want has been discarded. Calling the entry point (the reload thunk) is the easiest way to bring the segment back.

Once the segment has been loaded, we can use it as much as we want, as long as Windows doesn't discard it. But if we never call the segment's entry point again, Windows will think we've stopped using it — after all, it's the calls through the thunk that keep track of usage. Without periodic calls to the entry point, this segment will be one of the first to be discarded, no matter how much we've actually been using it.

Fortunately, Windows provides a mechanism to remind us to call the entry point periodically. On a regular basis (typically every fourth timer tick, or 4.5 times per second), Windows performs an "LRU sweep." One of the things Windows will do during the LRU sweep is to fill part of our segment table with zeros. The number of words set to zero is specified in *SegmentTable[1]*; the zero fill starts at *SegmentTable[2]*. In addition, *SegmentTable[1]* itself is also set to zero, which means nothing will be zero-filled again until it is reset to some value. This use of *SegmentTable[1]* suggests using a macro to give it the name *cwClear*.

Some applications use large amounts of read-only (constant) data. An example of this is Microsoft Excel

The idea is to set aside the first portion of the segment table for read-only data. At every LRU sweep, Windows will zero fill the segment numbers that were stored in there. When we try to access a segment number that has been zeroed, we will see an even number and conclude it was discarded. Then we call the segment's entry point to reload it, and the thunk will record the activity. Hopefully, this will prevent the segment from being discarded while it is still needed. The overhead of zero filling the table and calling the entry point is quite small compared with the time to reload a segment from disk.

Note that the *segtable* library, as written, is not set up for this type of use. The non discardable data segments, such as *segDgroup*, must be moved above the zero-fill area in *SegmentTable*. Because there are a fixed number of read-only data segments, they would probably each have their own fixed

(continued from page 60)
segment table entry. New access macros would be required that could deal with a segment that was not present.

Debugging Considerations

Microsoft considers the ideal environment for running Windows to be a 80386 computer with extended memory running 386MAX by Qualitas of Bethesda, Maryland. 386MAX puts the computer into Virtual 8086 Mode and manages memory by using the 386's paging mechanism. It provides three important benefits for Windows. First, it fully emulates LIM 4.0 expanded memory (EMS). Second, it performs the same function as the Windows program HIMEM.SYS, making available the first 64K of extended memory for use by Windows. Third, it allows TSR programs such as mouse and network drivers to be loaded out of the way of conventional memory — the base 640K memory space.

When Windows finds itself loaded into a computer with LIM 4.0 EMS, and there's a fair amount (like 256K) of conventional memory left, it will use "large frame" EMS. This means that the base 640K memory space becomes part of the EMS page frame. Windows can then swap different logical memory pages into the base 640K.

While this is a great way to run Windows, especially when running several large applications, it's not so good for debugging with Symdeb. Symdeb seems to get confused by the EMS swapping, and we've gotten some very strange results. Now we always disable 386MAX whenever we will be debugging a Windows program with Symdeb. On the other hand, CodeView for Windows is apparently so large that Windows doesn't use large frame EMS. CodeView is so big that it requires EMS to run, and it works fine with 386MAX.

While my comments about EMS apply generally to Windows debugging, there is a booby trap specific to working with a segment table. (Naturally we're telling you this because it happened to us.) Recall that, during Windows' LRU sweep, *cwClear* (*SegmentTable[1]*) is used as a count of words in the segment table to zero fill. Should this word get accidently set through a programming error, unbelievably strange results can occur. A random value stored in *cwClear* will zero out a random amount of *DGROUP*, possibly including your stack. What makes this bug so nasty is that the LRU sweep is driven by the timer tick interrupt, so the data gets wiped out without you ever seeing how. Even a 386 hardware breakpoint will not necessarily catch it. (In our experience, the hardware breakpoint caught this bug when debugging with a serial terminal, but not when using a monochrome monitor.)

Extensions

As written, the *segtable* library and associated macros assume that the segments in the table are always present in memory. This is guaranteed by the fact that none of the segments in the table are marked as discardable. Except for *DGROUP*, they are all allocated by *SegmentAlloc( )*, which does not set the *GMEM_DISCARDABLE* flag.

If the use of the segment table was expanded to include read-only segments as discussed above, then there would be discardable segments in the table. An even value in a table entry would signify that that segment had been discarded. More complicated access macros would be needed to account for this possibility and to provide the mechanism to reload the segment. The macros could take one of two approaches. The first method would be to always call a near function for each segment reference, and that function would test for an even entry and perform the reload if needed. The alternative is to make the test for an even entry in line, and call a function only when reloading is necessary. In fact, having both of these forms available might be handy so that the speed/size tradeoff can be made on a case-by-case basis. It is likely that read-only segments would be used only in special ways, so that many segment table references could still assume the segment was always present and use the original, more efficient macros.

We have been describing the whole idea of the segment table as being suitable for large applications with multiple segments of data. There is, however, a limit on how much data a Windows program can have. Being nondiscardable, the data must be present in memory at all times. This usually limits an application to not more than 300K under the best conditions. Large frame EMS does not increase this limit, but it does allow each of several applications running simultaneously to have about as much data space as if they were running alone.

The problem is the 640K limit on conventional memory, and one possible answer is EMS. Windows will allow individual applications to control the small (LIM 3.2-style) EMS frame, which provides four 16K portholes into the EMS space. It is completely up to the application to manage its expanded memory, using interrupt *67H* to access EMS functions.

One way to go about this is to integrate EMS management with the mem-

ory management functions of the *segtable* library. Any data segment of less than 16K is a candidate for allocation in EMS instead of using *GlobalAlloc( )*. *SegmentAlloc( )* could be modified to do this, putting the EMS segment into the segment table and returning a *pSeg*. In this way, the use of EMS becomes completely transparent to the rest of the application.

There is, however, a serious drawback. Because there is space for only four EMS pages in the page frame, we can't allocate more than four pages before we run out of places to put them. Of course, the whole point of EMS is that we can have many megabytes of data, but we only need to use a few pages at any one time. Some of the EMS pages we allocate for data will have to be mapped out of the page frame — becoming momentarily inaccessible — so that others can be mapped in when we need them.

Fortunately, the segment table mechanism provides a handy way to do this. *pSeg*s are the handle by which the application can refer to any chunk of memory, whether conventional, accessible EMS, or inaccessible EMS. If the *pSeg* points to an odd-numbered value in the segment table, then that segment is present; if it points to an even-numbered value, then it is not present. This is exactly the same rule that is used for read-only data segments.

To take this approach, the application's EMS manager must ensure that EMS segments are odd. Whenever it must change the EMS map, it will have to update the segment table. When a page is mapped out, its segment number in the table must be found and replaced with an even-numbered marker. This marker must represent sufficient information to make the page accessible again. For example, 1 byte of the marker could represent an index into a table that includes the EMS handle, while the other byte is the logical page number. Remember that only 15 bits are available, because the least significant bit must be zero.

The access macros must understand how to deal with segments that aren't present, using the same general techniques as they would for read-only segments. However, the segment is "reloaded" by calling the EMS manager, instead of by calling a Windows reload thunk. The application's memory manager will need to have some reasonable way to decide which logical page to map out when a different one must be mapped in. One approach would be to approximate the LRU algorithm by discarding the least-recently mapped-in page. Then when two different segments, say A and B, are needed at the same time, this can be ensured by the sequence access-A, access-B, access-A. The second access-A is required because the access-B might have caused A to get mapped out. This could happen only if A was already present at the start, so that the first access-A did nothing.

Microsoft's own Windows applications use all of the techniques discussed here

To support cases when more than two segments were needed at once, a locking mechanism could be used. This would be similar to Windows' *GlobalLock( )* and *GlobalUnlock( )*, except that it would be handled by the application's memory manager. A streamlined alternative to making function calls for locking would be to set aside one or more special locations in the segment table. The presence of the segment in a special location would tell the memory manager not to map it out.

If the computer has no (or not enough) EMS, we can still do something to handle large amounts of data. By using the segment table and some additional help from Windows, we can set up a virtual memory system — that is, disk swapping. The key is to allocate memory with the Windows function *GlobalAlloc( )* by using the flags *GMEM_DISCARDABLE* and *GMEM_NOTIFY*. This tells Windows that it can discard the memory if it needs to, but to ask permission first. When Windows notifies the application that it would like to discard a segment, we can write that segment to disk first, then stick a marker for that segment in the segment table. As with EMS, the marker will represent the information needed to reload the segment the next time it is accessed.

The function that Windows will call to ask permission to discard a segment is set by using *GlobalNotify( )*. This function is documented in the Windows 2.0 SDK update booklet, with additional information in the Windows 2.1 SDK update. The function we register with Windows in this manner could be declared as:

```
BOOL FAR PASCAL
        NotifyProc(HANDLE hmem);
```


FULL AT&T C++: ANNOUNCING VERSION ~~1.2~~ 2.0!

Guidelines announces its port of **version 2.0** of AT&T's C++ translator. As an object-oriented language, C++ includes: classes, **multiple inheritance**, member functions, constructors and destructors, data hiding, and data abstraction. Object-oriented means that C++ code is more readable, more reliable, and more reusable. And that means faster development, easier maintenance, and the ability to handle more complex projects. C++ is **Bell Labs' answer to Ada and Modula 2.** C++ will more than pay for itself in saved development time on your next project.

C++

from GUIDELINES for the IBM PC: $395

Requires IBM PC-AT or compatible with 512K plus 384K extended memory.
Note: C++ is a *translator*, and requires the use of Microsoft C 4.0 or later.

Here is what you get:

- The full AT&T v2.0 C++ translator with extended memory support.
- Libraries for stream I/O and complex math.
- C++ **Primer**, the definitive book on C++ version 2.0 by **Stanley B. Lippman**.
- Sample programs written in C++.
- Printed installation guide and documentation.
- **30-day money-back guarantee.**

NOW AVAILABLE FOR UNIX V/386 - $495

To Order:

Send check or purchase order to:

GUIDELINES SOFTWARE, INC.
P.O. Box 6368, Dept. DDJ
Moraga, CA 94570

To order with VISA or MC, phone (415) 376-5527. (California residents add sales tax.)

C++ was ported by GUIDELINES under license from AT&T.
Call or write for a free C++ information package.

CIRCLE NO. 351 ON READER SERVICE CARD

(continued from page 63)
The argument is supposed to be the handle of the segment being discarded. However, the Windows 2.1 SDK update says that in Version 2.03, it was actually the segment number, not the handle. This can be straightened out for both versions by calling *GlobalHandle( )*, which can take either the handle or segment number as its argument, and will return them both, as mentioned earlier.

NotifyProc( ) is a function in the application, but it must be in a fixed code segment. It will be called for each segment Windows would like to discard. If the application wants the segment locked, the function can return a false (zero) value and Windows will not discard it. The locking protocols could be the same as we suggested for EMS: Adding lock and unlock functions, and/or reserving special locations in the segment table. If the segment isn't locked, *NotifyProc( )* can write it to a disk file that has already been created for that purpose. Then it returns true and Windows will reclaim the space.

Any of these extensions — read-only data, EMS, disk swapping — may be combined. Using any one of them requires handling the case of segments that are not currently accessible. Once this jump has been made, the others can be added with little or no additional change to the main body of the application. Microsoft's own Windows applications use all of the techniques discussed here (a great deal of time was spent using Symdeb on Excel in preparing this article). While we haven't covered all of the procedures in detail, these ideas can be used to build Windows applications with virtually unlimited data capacity.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 106.)

Vote for your favorite feature/article.
Circle Reader Service **No. 5.**

Object-Oriented Programming in Assembly Language

OOP applies equally well to assembly language and high-level language programs

Randall L. Hyde

One of the promises of the object-oriented paradigm is that it will reduce program complexity and implementation effort for many different types of programs. Object-oriented programming, however, is no panacea. It is a technique, like recursion, that you can apply in certain cases to reduce programming effort. While there are certain types of programs whose object-oriented implementation is better, examples abound where object-oriented programming systems (OOPS) buy you nothing. Nonetheless, object-oriented programming techniques are a valuable tool to have in one's war chest.

OOPS are nothing new. They have been around since the late 1960s. Yet the object-oriented paradigm was languishing until Object Pascal and C++ began generating mainstream interest. The success of these languages demonstrates that OOP is not the domain of a few esoteric programming languages. Rather, object-oriented programming is applicable to almost any programming language.

Still, assembly language may not seem like the place to apply the object-oriented programming paradigm. But keep in mind that people were saying the same thing about Pascal and C five years ago.

What does an object-oriented assembly language program look like? A better question to ask is, "What is the essence of an object-oriented program, and how does one capture it within an assembly language program?" Once you strip away the gloss and notation convenience provided by languages such as C++, you'll find that the two main features of an object-oriented program are polymorphism and inheritance.

Polymorphism takes two basic forms in most programming languages: static and dynamic. The general idea, however, is the same. You call different subroutines by the same name. Static polymorphism provides notational convenience in the form of operator/function overloading in languages such as C++. Static polymorphism uses the parameter list, along with the routine's name (together they form the routine's signature), to determine which routine to call. For example, consider the C routines:

```
CmplxAddCC(C1, C2, C3);
                    /*C1=C2+C3;*/
CmplxAddCR(C1, C2, R1);
          /*C1=C2+ToCmplx(R1);*/
CmplxAddRC(C1, R1, C2);
          /*C1=ToCmplx(R1)+C2;*/
```

In C++ you could write:

```
CmplxAdd(C1, C2, C3);
CmplxAdd(C1, C2, R1);
CmplxAdd(C1, R1, C2);
```

and the C++ compiler would figure out whether to call *CmplxAddCC*, *CmplxAddCR*, or *CmplxAddRC*. (Actually, you could overload C++'s "+" operator and use the three forms *C1=C2+C3;*, *C1=C2+R1;*, or *C1=R1+C2;*, but the example above would be still valid.)

Static overloading, while convenient, does not add any power to the language. The calls to *CmplxAdd* call three different routines. *CmplxAdd-(C1,C2,C3)* calls *CmplxAddCC*, *CmplxAdd (C1,C2,R)* calls *CmplxAddCR*, and *CmplxAdd-(C1,R,C2)* calls *CmplxAddRC*. The C++ compiler determines which routine this code will call at compile time. Static polymorphism is a mechanism that lets the compiler choose one of several different routines to call depending upon the calling signature.

Sometimes you may want to use the same signature to call different routines. For example, suppose you have a class *shape* in which there are three graphical objects: circles, rectangles, and triangles. If you have an arbitrary object of type *shape*, the compiler cannot determine which *DRAW* routine to call. The program determines this at run time. This allows a single call to draw circles, rectangles, triangles at run time with the same machine instructions. This is dynamic polymorphism — determining at run time which routine to call. C++ uses virtual functions and Ob-

Randy is the designer of numerous hardware and software projects, including assemblers for a variety of systems. In addition to consulting, he is currently a part-time instructor in computer science at California Polytechnic University in Pomona and at UC Riverside. He can be contacted at 9570 Calle La Cuesta, Riverside, CA 92503.

(continued from page 66)
ject Pascal uses override procedures and functions to implement dynamic polymorphism.

Inheritance lets you build up data structures as supersets of existing data structures. This provides a mechanism whereby you can generalize data types, allowing you to handle various objects regardless of their actual type. This lets you define such diverse shapes as circles, rectangles, and triangles and treat them as compatible structures.

Implementing Classes and Inheritance

Because structures and classes are closely related, it may be instructive to look at the implementation of structures before looking at classes. Consider *S*, a variable of the type in Example 1. Somewhere in memory the compiler needs to generate storage for the fields of *S*. Traditionally, compilers allocate these fields contiguously (see Figure 1). Indeed, Microsoft's assembler (MASM) allows you to declare structures in a similar fashion, as shown in Example 2. If *S* provides the base address of this structure, *S+0* is the address of *S.i*, *S+2* is the address of *S.j*, and *S+4* is the address of *S.c*.

Now consider the case of a pair of C++ classes (*Sc* and *Tc*) in Example 3. Pointers to objects (*pS* and *pT*) may point at an object of the prescribed type or to an object that is a descendant of the pointer's base class. For example, *pS* can point at an object of type *Sc* or at an object of type *Tc*. Remember, accessing **pS.j* is equivalent to *(int) *(pS+2)*, so if *pS* points at an object of type *Tc*, the *j* field must also appear at offset two within the structure. For inheritance to work properly, the common fields must appear at the same offset within the structure (see Figure 2).

Additional fields in the subclass often appear after the fields in the parent class, so most compilers implement class *Tc* as in Example 4. Note that the offsets to *i*, *j*, and *c* are the same for both *Sc* and *Tc*.

When I first began exploring how to implement inheritance in assembly, I got the bright idea of using macros inside structure definitions to handle the problem of inheritance. Briefly, I wanted to implement *Sc* and *Tc* as in Example 5. Unfortunately, MASM doesn't allow you to expand macros or strucs inside a structure. Disappointed, I tried the brute force way to implement *Sc* and *Tc*, as illustrated in Example 6.

Unfortunately, I'd forgotten that MASM doesn't treat these symbols as part of the structure. Names such as *i*, *j*, *c*, and so on must be unique in the program. As you can plainly see in Example 6, I declared *i* twice, and the assembler gave me a "redefinition of symbol" error. Almost ready to give up, I tried the method in Example 7.

MASM simply equates the field names to the offsets within the structure. So it equates *i* to zero, *j* to two, and so on. MASM does not associate *i* with labels of type *Sc*. You can use the symbols *T.j* and *S.j* in your program. Because the "." operator behaves like the "+" operator, *T.j* is just like *T+2*.

For *Tc* to inherit the fields of *Sc*, all we have to do is reserve enough space at the beginning of the *Tc* structure for

Example 1: *The variable* S

```
struct
{
        int i;
        int j;
        char *c;
} S;
```

Example 2: *Declaring structures in MASM*

```
SType   struc
i       dw      ?
j       dw      ?
c       dd      ?
SType   ends
```

Example 3: *C++ classes*

```
class Sc
{
          int i;
          int j;
          char *c;
};

Sc *pS;
Tc *pT;
```

```
Tclass Tc:Sc
{
           int k;
           char *d;
};
```

Example 4: *The way most compilers implement a class like* Tc

```
struct Tc
{
      int i;
      int j;
      char *c;
      int k;
      char *d
};
```

```
Tc    struc
i     dw      ?
j     dw      ?
c     dd      ?
k     dw      ?
d     dd      ?
Tc    ends
```

Example 5: *One approach to implementing* Sc *and* Tc

```
ScItems macro
i       dw      ?
j       dw      ?
c       dd      ?
endm

;
TcItems   macro
          ScItems
k         dw      ?
d         dd      ?
          endm

;
Sc        struc
          ScItems
Sc        ends
;
Tc        struc
          TcItems
Tc        ends
```

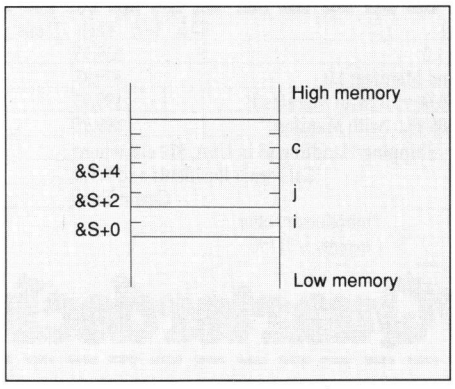


Figure 1: *Storage allocation for* S

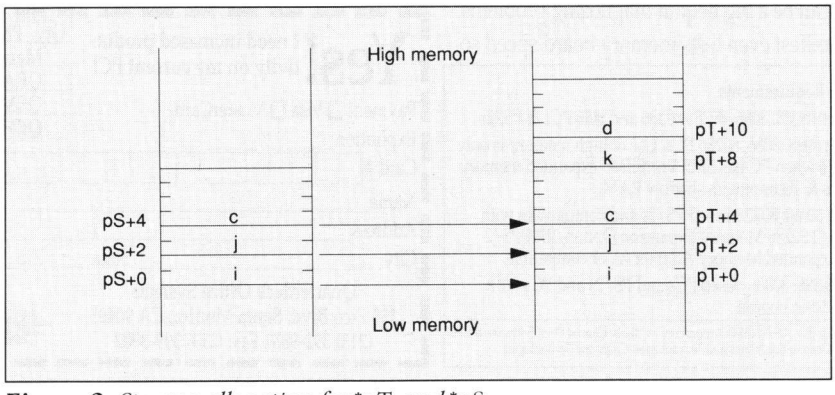


Figure 2: *Storage allocation for* *pT *and* *pS

(continued from page 68)
each of the fields of *Sc*. Above, I stuck in the two *DW* and the *DD* pseudo-opcodes to reserve space for the *i*, *j*, and *c* fields. This technique might get inconvenient if the number of inherited fields is large. The code in Example 8 solves this problem.

The first *DB* pseudoopcode in *Tc* reserves the necessary space for the fields *Tc* inherits from *Sc*. Likewise, *Uc* (which is a subclass of *Tc*) reserves space at the beginning of the structure for the fields inherited from *Tc* and *Sc*. The code in Example 8 works great if you don't need to initialize any of the fields inherited from *Sc*; if you need to initialize some fields, you'll have to use the brute force method and redeclare space for each field.

Methods

The earlier paragraphs discuss how to implement objects whose fields are all variables. What happens when you introduce methods? If you're not overloading a method, you can treat it in the same manner as any other assembly language procedure and call it directly. If you are overloading a method, you must call it indirectly via a pointer within the object.

Consider the C++ class declaration in Example 9. The assembly code implementing this class is shown in Example 10. To call *S.geti*, you would use the 8086 instruction: *CALL S.geti*.

Because *S.geti* is a double word memory variable, the *CALL* instruction will call the procedure *S.geti*, which points at *Sc_geti*. The fact that we're calling the methods indirectly will be useful when we look at overloading a little later.

THIS

Suppose we have three instances of class *Sc*, say *S1*, *S2*, and *S3* declared in assembly language as follows:

```
S1  Sc
S2  Sc
S3  Sc
```

S1.geti, *S2.geti*, and *S3.geti* all call the same procedure (call it *Sc_geti*). How does *Sc_geti* differentiate between *S1.i*, *S2.i*, and *S3.i*? In object-oriented languages such as Object Pascal and C++, the compiler automatically passes a special parameter named *this* to the method. *this* always points at the object through which you've invoked the method. When you execute *S1.geti*, the compiler passes the address of *S1* in *this* to *geti*. Likewise, the compiler passes the address of *S2* in *this* when you call *S2.geti*.

You can pass *this* to a method just as any other parameter. Because the most efficient way of passing parameters is in the 8086's registers, I've adopted the convention of passing *this* in the ES:BX registers. The *Sc_geti* method would look something like Example 11 (assuming we're returning *i* in the AX register). This example demonstrates a major problem with object-oriented programming — it is very inefficient. To load *S1.i* into AX, see Example 12. This requires six instructions where, logically, you should only need one (*mov ax, S1.i*). Welcome to the wonderful world of object-oriented programming! Yet circumventing all this overhead by loading *S1.i* directly into AX will eliminate the benefits of object-oriented programming.

Actually, this isn't as bad as it looks. A good part of the time ES:BX will already be pointing at the object you want to access. Nevertheless, the call and return are considerable overhead just to load the AX register with a word value. Stroustrup anticipated this problem when designing C++ and he solved it by providing inline functions (a.k.a.

Example 6: *The brute force method of implementing* Sc *and* Tc

```
Sc      struc
i       dw      ?
j       dw      ?
c       dd      ?
Sc      ends

Tc      struc
i       dw      ?
j       dw      ?
c       dd      ?
k       dw      ?
d       dd      ?
Tc      ends
```

Example 7: *Yet another attempt at implementing* Sc *and* Tc

```
Sc      struc
i       dw      ?
j       dw      ?
c       dd      ?
Sc      ends

Tc      struc
        dw      ?
        dw      ?
        dd      ?
k       dw      ?
d       dd      ?
Tc      ends

S       Sc
T       Tc
```

Example 8: *The solution to implementing* Sc *and* Tc

```
Sc      struc
i       dw      ?
j       dw      ?
c       dd      ?
Sc      ends

Tc      struc
        db      (size Sc) dup (?)
k       dw      ?
d       dd      ?
Tc      ends

Uc      struc
        db      (size Tc) dup (?)
e       dw      ?
Uc      ends

S       Sc
T       Tc
U       Uc
```

Example 9: *A C++ class declaration*

```
class Sc
{
      int i,j;
      char *c;

      public:

      int geti() {return i};    /* Ignore the fact that C++ */
      int getj() {return j};    /* would implement these    */
      void seti(x) int x; {i = x;};  /* methods in-line.    */
      void setj(x) int x; {j = x;};
};
```

Example 10: *Assembly code for implementing the code in Example 9*

```
Sc      struc
i       dw      ?
j       dw      ?
c       dd      ?
geti    dd      Sc_geti
getj    dd      Sc_getj
seti    dd      Sc_seti
setj    dd      Sc_setj
Sc      ends

S       Sc
```

(continued from page 70)
macros). We can use this same technique in assembly language to improve efficiency as Example 13 illustrates. This code snippet demonstrates another convention I adhere to: I make macros for all method calls, even those that are actual calls. This lets me use a consistent calling format for all methods, whether they are actual subroutines or are expanded in-line.

There is one major drawback to expanding a procedure inline; you cannot overload procedures (C++'s inline functions suffer from this as well. You cannot have an inline virtual function). Therefore, you should only use this technique for those particular methods that you will never need to overload. Fortunately, the macro implementation makes it easy to switch to a call later if you need to overload the procedure. Just substitute a call for the inline code inside the macro.

Polymorphism and Overloading

Overloaded procedures allow the "same" method to perform different operations, depending upon the object passed to the method. Consider the class definitions in Example 14. *Rect* and *Circle* are types derived from *Shape*. If ES:BX points at a generic shape (that is, ES:BX points at an object of type *Shape*, *Rect*, or *Circle*) then *CALL_THIS.Draw* will call *Shape_Draw*, *Rect_Draw*, or *Circle_Draw*, depending upon where ES:BX points. This lets you write generic code that needn't know the particular details of the shape it's drawing. The object itself knows how to draw itself via the pointer to the specific draw routine.

Allocation of Objects

High-level object-oriented languages such as Object Pascal and C++ tend to hide many of the allocation details from you. In assembly language, naturally, the programmer has to handle all of the allocation details. Although a complete discussion of dynamic allocation of objects is beyond the scope of this article, the subject is so pervasive that it warrants a brief mention.

Static allocation of an object in assembly language is quite simple. If you have the *shape* class definitions (*shape*, *rect*, and *circle*) mentioned earlier, you can easily declare variables of these types using declarations of the form:

```
MyRect      rect
MyCircle    circle
MyShape     shape
```

This automatically fills in the *DRAW* field for these variables (the linker/loader fills in such addresses when it loads the program into memory). What happens if you are dynamically allocating storage for an object? Assume we have a routine, *alloc*, to which we pass a byte count in CX, and it returns a pointer to a block of memory that size in ES:BX. Now suppose we allocate a rectangle with the code in Example 15. *Alloc* will not be smart enough to fill in the pointer to the *rect.DRAW* routine. This is something we'll have to do ourselves. This requires the four instructions in Example 16.

Eight instructions may not seem like a lot to create a simple object. Keep in mind, however, that our simple *shape* object only has one overridden method. If there were a dozen methods, you would need 52 instructions. Clearly, a *CREATE* procedure begins to make a lot of sense. Each subclass (*shape*, *rect*, and *circle*) will need its own *CREATE* method. *CREATE* is not a method you normally overload, because during the creation process you know exactly the type of object you're creating. By convention, the *CREATE* methods I write always allocate the appropriate amount of storage, initialize any important fields, and then return a pointer to the new object in ES:BX. The code in Example 17 provides an example, using the *rect*

Example 11: *The* Sc_geti *method*

```
_THIS       equ     es:[bx]
Sc_geti     proc    far
            mov     ax, _THIS.i
            ret
Sc_geti     endp
```

Example 12: *Loading* S1.i *into AX*

```
mov     bx, seg S1
mov     es, bx
mov     bx, offset S1
call    S1.geti         ;Assuming S1 is in the data seg
```

Example 13: *Improving efficiency*

```
;
; Inline expansion of geti to improve efficiency:
;
_Geti               macro
                    mov             ax, _THIS.i
                    endm
;
; Perform actual call to routines which are too big to
; expand in-line in our code:
;
_Printi             macro
                    call            _THIS.Printi
                    endm
                      .
                      .
                      .
                    _Geti           ;Get i into AX.
                      .
                      .
                      .
                    _Printi         ;Call Printi routine.
```

Example 14: *Typical class definitions*

```
Shape       struc
ulx         dw      ?       ;Upper left X coordinate
uly         dw      ?       ;Upper left Y coordinate
lrx         dw      ?       ;Lower right X coordinate
lry         dw      ?       ;Lower right Y coordinate
Draw        dd      Shape_Draw  ;Default (overridden) DRAW routine
Shape       ends
;
Rect        struc
            dw      4 dup (?)   ;Reserve space for coordinates
            dd      Rect_Draw   ;Draw a rectangle
Rect        ends
;
Circle      struc
            dw      4 dup (?)   ;Reserve space for coordinates
            dd      Circle_Draw ;Draw a circle
Circle      ends
```

and *circle* types. To manipulate these objects, we need only load the appropriate pointer into ES:BX and access the appropriate fields or call the appropriate methods via *this*.

Other Conventions

While writing object-oriented programs in assembly language, I've found certain guidelines helpful in the initial design phases (that is, before having to take efficiency into consideration). Most of these guidelines are widely accepted object-oriented practices; others pertain mainly to assembly language. Here are the major ones I'm using:

- Try to use dynamic allocation for objects wherever possible. In the best object-oriented programs, instances of an object appear and disappear throughout the program. Rarely will a single instance exist throughout the execution of a program. Because an object's methods always reference fields of an object indirectly, there is little benefit to statically allocated objects. Converting a statically allocated object to a dynamically allocated one later on is messy. Get it right the first time!
- Avoid accessing the individual variables (fields) within an object. Write methods that store values into these fields and retrieve values from them. This information-hiding technique is well proven in OOP and isn't particularly worthy of further discussion.
- Overload as many methods as possible. *CREATE* is probably the only method you shouldn't overload. Access methods, which provide access to the fields of the outermost class, might be another candidate for direct access. But the loss of generality for a small increase in efficiency is rarely worth it.
- Always use macros to call methods, especially those you're not calling indirectly. This provides a consistent calling mechanism for methods and lets you easily overload methods you choose to implement inline or without overloading. This applies equally well to accessing fields in an object.
- As a bare minimum, each class should have the following methods: *CREATE*, *DISPOSE*, *COPY*, and a set of access methods for each of the fields. *COPY* should copy the contents of one instance variable's fields to another variable.

Naturally, these are just guidelines, not rules etched in stone. But a certain amount of discipline early in a project helps prevent considerable kludging later on.

An Example

The example in Listing One (page 110) is a program that adds, subtracts, and compares signed binary integers, unsigned binary integers, and BCD values. While not a complete example (it's missing several important methods such as *CREATE*, *PRINT*, *DISPOSE*, and so on) it demonstrates the flavor of object-oriented programming in assembly language.

What About Your Programs?

Object-oriented programming is a concept that can reduce the time you spend developing certain classes of programs. The OOP concept applies equally well to assembly language and high-level language programs. The only drawback is that you don't have a large library of classes to build upon. Of course, these same problems exist for Object Pascal and C++ users. Time will solve this problem for those languages as users begin developing reusable modules for both, which is all that is preventing object-oriented assembly language from taking off. Perhaps someday you will be able to buy off-the-shelf object-oriented assembly language libraries; until then, you'll have to write your own. Even so, the tricks and techniques of object-oriented programming are well worth considering for your next assembly language project.

DDJ

(Listing begins on page 110.)

Vote for your favorite feature/article.
Circle Reader Service **No.6.**

Example 15: *Code to allocate a rectangle*

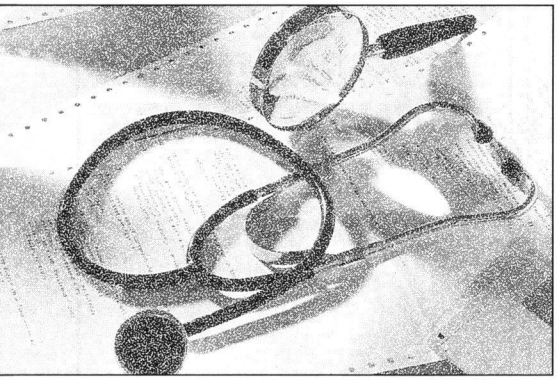
```
mov     cx, size rect
call    alloc
mov     word ptr MyRectPtr, bx
mov     word ptr MyRectPtr+2, es
```

Example 16: *Filling in the pointer to the* rect.DRAW *routine*

```
mov     ax, offset rectDRAW
mov     _this.DRAW, ax
mov     ax, seg rectDRAW
mov     _this.DRAW+2, ax
```

Example 17: *Code for example using the* rect *and* circle *types*

```
mov     cx, size circle
call    CreateCircle
mov     word ptr CircVarPtr, bx
mov     word ptr CircVarPtr+2, es
;
mov     cx, size rect
call    CreateRect
mov     word ptr RectVarPtr, bx
mov     word ptr RectVarPtr+2, es
```



Inside Watcom C 7.0/386

32-bit code can speed up your programs on an already quick machine

Andrew Schulman

Over two years ago, the cover of the July 1987 issue of *Dr. Dobb's* carried the title "386 Development Tools Within Your Lifetime" a photograph of a skeleton that rotted away in front of its computer while waiting for decent 386 tools, which summed up everyone's feelings about programming for the Intel 80386 microprocessor.

Things have improved a great deal since that issue. Watcom C7.0/386, for instance, produces 32-bit code (such as *MOV EAX, 12345678h*, and *MOV FS:[EAX], ESI*) while staying keyword and library compatible with the de facto 16-bit industry standard, Microsoft C 5.1 (MSC51). Even weird low-level routines such as *intdosx( ), _dos_setvect( ), _dos_keep( )*, and *_chain_intr( )* do the right thing in 32-bit protected mode.

Of course, Watcom C7.0/386 (WAT386) has many of the same features as Watcom's 16-bit C compiler (see "Examining Room," *DDJ* September 1989). This includes Watcom's famous register-based parameter passing. Many of Watcom's innovations involve the reduction, and sometimes elimination, of function call overhead. Any block of code that takes input from registers and puts output into registers is effectively a functional object, and WAT386 takes advantage of this fact in several places, including the nifty *#pragma aux* feature.

Andrew is a software engineer in Cambridge, Mass., working on CD-ROM network applications, and is also a contributing editor for DDJ. *He can be reached at 32 Andrew St., Cambridge, MA 02139.*

Buying In

WAT386 produces very different code from either Microsoft C or Turbo C (neither of which has an option to generate 386 instructions, much less 32-bit code). Yet, this compiler will fit seamlessly into your current work habits. Unlike MetaWare's High C 386 compiler, using WAT386 does not produce "culture shock."

Still, all is not rosy. It will cost you over $1000 in software to get into 386 development. WAT386, like High C, costs $895, and you will also need a 32-bit DOS extender, like the industry-standard Phar Lap 386 toolkit, which costs $495.

Further, the new Watcom C7.0/386 compiler is just that — new. While writing this review, I found a number of bugs in the compiler and its standard library. Watcom was undoubtedly under pressure from its major client, Novell, to get the 386 compiler out the door. By the time you read this review, though, a second, more stable, release of WAT386 should be available.

Primarily because of its newness, WAT386 in some ways is not as good a product as MetaWare's High C 386, which has been around for two and a half years. Still, there is value in WAT386. For many PC programmers, this will be a much easier product to use than MetaWare's High C. WAT386's Microsoft compatibility is very important. On the other hand, the next release (1.6) of High C 386, in addition to many other changes, is scheduled to have what a MetaWare press release calls "86% compatibility with Microsoft's C libraries."

32 Bits!

WAT386 generates code for 32-bit protected mode. Thus, *sizeof(int)* and *sizeof(unsigned)* are each 4 bytes, not 2 bytes. Likewise, *sizeof(void *)* is 4 bytes. Note that *sizeof(void near *)* is also 4 bytes.

The all-important ANSI C identifier *size_t*, which is the unsigned type of the result of the *sizeof( )* operator and the type used by function parameters that accept the size of an object, is also 4 bytes (*typedef unsigned size_t*).

C standard library functions such as *malloc( ), fwrite( )*, and *strncpy( )* all take *size_t* parameters, and *strlen( )* returns a *size_t*. These standard library functions deal in quantities between *0* and *UINT_MAX*. In the 16-bit code generated by PC compilers like MSC51, *UINT_MAX* is *0xFFFF* (65,535), yielding the familiar 64K limit on PC array lengths, string lengths, malloc blocks, and so on.

But in 32-bit code, *UINT_MAX* is *0xFFFFFFFF*, or 4,294,967,295 — the magical upper "limit" of 4 gigabytes! In the native mode of the 386, this is the upper bound set on array lengths, string lengths, and malloc blocks. Effectively, no limit at all.

The Environment

If *fwrite( )* can write 4 gigabytes at a time (which might be handy if you're working with CD-ROM or some other form of mass optical storage), how can it possibly work with MS-DOS? DOS is a 16-bit operating system. (So is OS/2.) The DOS *Write* function (*INT 21, function 40H*), which *fwrite( )* must eventually call, expects the number of bytes

Programmers Wholesaler™

Attention!

Corporate Accounts
Resellers
Programmers

Check our values!

	LIST	1-2	3+
BASIC			
Turbo Basic	100	67	64
QuickBASIC	99	67	64
Basic Dev. Sys. 7.0	495	329	321
C LANGUAGE - COMPILERS			
Lattice C - 6.0	250	156	143
Microsoft C 5.1	450	287	283
Microsoft Quick C	99	67	64
Turbo C by Borland	150	98	94
DATABASE MANAGEMENT			
Clarion	695	399	379
Paradox 3.0	725	489	479
DBASE			
Clipper Summer '87	695	429	419
dBASE IV	795	489	479
FoxBASE + 2.1	395	209	199
DBASE TOOLS			
Clear+ for dBASE	200	149	139
dBRIEF w/BRIEF	285	Save	Save
dSalvage	100	83	79
R&R Relational Reportwriter	149	99	93
EDITORS			
BRIEF	199	Save	Save
Epsilon	195	139	109
FORTRAN			
MS FORTRAN	450	299	289
OBJECT-ORIENTED			
Smalltalk/V	100	59	54
Zortech C++	199	Call	Call
OTHER PRODUCTS			
Carbon Copy Plus	195	115	104
HEADROOM by Helix	130	85	79
Norton Utilities Advanced	150	89	87
PC Tools Deluxe	129	85	79
Remote2	195	104	99
SPREADSHEETS			
1-2-3	495	299	289
Excel	495	339	329
TEXT SCREENS ADDONS			
C Worthy w/Forms	295	Save	Save
Greenleaf DataWindows	395	249	239
Vermont Views	395	319	299
WORD PROCESSING			
Sprint	200	134	129
WordPerfect	495	239	234

Prices subject to change without notice. "DD390W"

Programmers Wholesaler™

800-228-3736

CANADA
800-344-2495
FAX 617-740-1892

South Shore Park
P.O. Box 534
Accord, MA 02018
Hours: M-F 8:30-5

CIRCLE NO. 199 ON READER SERVICE CARD

(continued from page 74)

to write in the 16-bit CX register. The maximum is 64K. How can WAT386, or any 32-bit C compiler for DOS, produce code that's compatible with 16-bit DOS?

The answer is that 386 C compilers (for DOS) produce code to be run under a 32-bit DOS extender. Programs such as Phar Lap's 386|DOS-Extender and Eclipse Computer Solutions' OS/386 do not replace DOS. Instead, they (almost invisibly) manage the interface between 16-bit real-mode DOS and your 32-bit protected-mode program.

In the example of *fwrite( )*, the 32-bit code produced by WAT386 or High C (which MetaWare actually calls "High C for MS-DOS/386") continues to call *INT 21, function 40H*. But now, the number of bytes to write goes into the full 32-bit ECX register rather than the 16-bit CX register.

A DOS extender takes over *INT 21* (as well as other software interrupts like *INT 10*, *INT 16*, and so on), handles some functions itself, and passes others on to DOS. A program running under a 32-bit DOS extender is effectively running under "MS-DOS/386," because, for example, a call to write 640K is really going to write 640K. The DOS extender will invisibly break this up into multiple calls to the "real" *INT 21, function 40H*.

Another interesting example is *malloc( )*. If your 386 computer came with 4 gigabytes of memory, you could grab it all with a single call to *malloc( )*. As in 16-bit real-mode C compilers, the C memory manager eventually calls *INT 21, function 48* (allocate memory). Here, however, the DOS extender provides a complete replacement, not a front end, for the DOS routine. There is one difference between Phar Lap and Eclipse: 386|DOS-Extender expects in EBX the number of 4K pages to allocate, where OS/386 more closely mimics DOS, expecting the number of 16-byte paragraphs. The WAT386 standard library detects which DOS extender it is running under and allocates memory appropriately.

By default, WAT386 produces code to be run under Phar Lap Software's 386|DOS-Extender. The Phar Lap toolkit (DOS extender, linker, assembler, and debugger) must be purchased separately, however.

Oddly, you don't need a 386 machine or a DOS extender to run the WAT386 compiler. By the time you read this review, Watcom should be shipping a 32-bit protected-mode version of the compiler. In the version I reviewed, however, all compiler components were 16-bit real-mode programs. To avoid "Not enough memory to fully optimize procedure" warnings, I had to specify that the compiler use a large-model version of the code generator. Pretty crazy for a 386 development system!

Presumably, if your customers had 386s but you didn't (which is probably the exact opposite of the real situation), you could use these 16-bit tools to generate 386 code on your AT.

Programs compiled with WAT386 and linked with Phar Lap's 386|LINK will only run on 386-based machines. To sell such programs, and to acquire a program that will "bind" the DOS extender into the executable so that your customers don't need to know anything about the DOS extender, you must acquire a redistribution package from Phar Lap. This costs an extra $1000 for unlimited distribution.

So the entrance fee for 386 development is still pretty steep. What do you get in return? A lot: Code that runs several times faster than 16-bit code; the elimination of 64K limits on array sizes or function parameters; and the elimination of the 640K boundary, allowing you to use all physical memory in the machine.

Note that this "MS-DOS/386" gives you big memory, but not virtual memory (VM). This is an important difference from OS/2. However, a VM manager (386|VMM) is available for $295 from Phar Lap, and WAT386 code, like High C code, runs without change under 386|VMM.

WAT386 code runs under one other environment: Novell's new 32-bit network operating system, NetWare 386 (see the accompanying box).

The Code

How can a 32-bit C compiler such as WAT386 produce code that runs several times faster than 16-bit code run on the same machine? Consider the following two lines of code:

```
extern char*Env;
char *p=Env
```

Compiling under the "large model" (which is what most commercial PC software uses), any 16-bit C compiler, including Watcom's non-386 compiler, produce code something like that shown in the first portion of Example 1, in which the 4-byte *far* pointers are transferred piecemeal from one location to another.

Because *mov mem, reg* takes 2 clock cycles on a 386 and *mov reg, mem* takes 4 cycles regardless of whether the compiler uses the 8-bit (AL), 16-bit (AX), or 32-bit (EAX) form of the register, this takes (2*2) + (3*4) = 16 cycles. In con-

(continued on page 79)

(continued from page 76)
trast, the 32-bit equivalent takes 2 + 4 = 6 cycles (shown in the second portion of Example 1).

The 32-bit code is similar to the code that would be generated by a 16-bit compiler working with 2-byte *near* pointers:

```
mov ax, _Env
mov word ptr _p, ax
```

In fact, "flat model" 32-bit code and "tiny model" 16-bit code are very similar. The only difference is that the 16-bit code can handle quantities up to

Watcom and Novell

In addition to producing code for Phar Lap's 386|DOS-Extender and, with some difficulty, for Eclipse's OS/386, the 32-bit Watcom C compiler also works with Novell's new network operating system, NetWare 386. In fact, Watcom C7.0/386 is being repackaged by Novell as its C Network Compiler/386. (This is the subject of Novell's strange "See Dick and Jane" ads.)

NetWare 386 is a 32-bit operating system, and this allows for several performance leaps over the existing 286-based NetWare. Instead of the current limit of 100 users per file server, which is dictated by the single 64K data segment available in "medium model" (used in 286-based NetWare), the new NetWare 386 allows 250 simultaneous users per file server. Likewise, Novell claims that network throughput is two to three times greater than its already zippy throughput figures.

In NetWare 386, the lack of segmentation in "flat model" is taken to its logical (but scary) extreme — no memory protection. Novell baldly states that, "There is no memory or other application-level protection: All applications and device drivers run in kernel mode" (*NetWare Technical Journal*, July 1989).

When used with NetWare 386, the Watcom C compiler produces server applications — programs that run in file-server memory (the so-called "file server" thus becomes a generic server). These server applications are called "NetWare Loadable Modules," or NLMs, and are somewhat like value-added processes (VAPs) in pre-386 NetWare; except unlike VAPs, NLMs can be loaded or unloaded at any time, without taking down the file server. NLMs, in fact, are dynamic-link libraries and, in addition to providing services to clients on the network, can provide functions to be called by other NLMs.

For instance, when calling a C standard library such as *open( )* from an NLM, you are actually calling a routine in CLIB.NLM, which is the C standard library provided as a dynamic-link library. The code for *open( )* is not linked into your executable.

To produce such an NLM, use the NLMLINK provided by Novell rather than the Phar Lap linker. Similar to the OS/2 linker, NLMLINK requires a .DEF file with import statements. The module produced by the Novell linker essentially contains unresolved externals that are resolved when the NLM is loaded into file server memory (either by invoking the LOAD command at the file server console, or by spawning one NLM from within another).

The library included with C Network Compiler/386 includes many functions not available in the standard Watcom library. Naturally, functions are provided to support network communications with Novell's IPX and SPX. The Btrieve data management library is provided as BTRIEVE.NLM. The Novell library includes functions (for example, *TestAndSetBit( )* and *BitScan( )*) to interface to the 386-bit test instructions.

Network servers are inherently multitasking (multiple operations must be in progress simultaneously on behalf of multiple clients), so the library contains functions for "execution threads," such as *BeginThread( ), EnterCritSec( ), ExitCritSec( ), SuspendThread( )*, and so on. There are also functions to manage semaphores and queues.

While this part of the Novell API seems modeled on OS/2, it is important to note that NetWare 386 uses non-preemptive multitasking. Inside a "big job," it is therefore necessary to call a routine such as *delay( )* or *ThreadSwitch( )* so that other threads are not starved.

The library that Watcom provided for Novell contains a few modifications to support multiple threads. Global variables such as *errno* are in fact allocated on a prethread basis. Static data such as used by the notorious *strtok( )* function is also handled differently than in a single-threaded library. No new keywords (such as *private*, used in Lattice C 6.0 for OS/2) have been added, however. — A.S.

64K, whereas the 32-bit code can handle quantities up to 4 gigabytes.

Right now, WAT386 supports flat model and small model. In the flat memory model, the application's code and data must total less than 4 gigabytes in size. In the small memory model, your code and data are each "limited" to 4 gigabytes. By default, WAT386 uses the flat model. When linking with the Lahey linker (LINK-EM/32) provided with OS/386, you must compile with the small model.

Because an offset into a segment is 4 bytes while the segment registers are still 2 bytes, *sizeof(void far *)* is 6 bytes (an FWORD, not a DWORD). But because a *near* pointer is a 4-byte quantity, you almost never have to deal with *far* pointers. When a segment takes a 4-byte offset, even the most sloppily written, bloated program in the world should do fine with the flat model. Once loaded, DS and CS stay constant. Effectively, this is a linear address space.

Real-World Benchmarks

Interpreters are better for benchmarking compilers than the tiny programs that are usually used. Such benchmarks usually involve a fair amount of source code. The C source code for several interpreters is readily available, and to execute one line in the interpreted language, the interpreter needs to crunch through a lot of C code.

In the remainder of this review, I'll describe using WAT386 (and MetaWare High C) to port a larger program to the 386: ISETL (Interactive Set Language), written in C by Gary Levin (Dept. of Mathematics and Computer Science, Clarkson University, Potsdam, N.Y.). ISETL is an interpreter for working with sets, tuples, propositions, several different types of functional objects, matrices, and other constructs useful for studying the mathematical foundations of computer science. It is described in the book *Learning Discrete Mathematics with ISETL* by Nancy Baxter, Ed Dubinsky, and Gary Levin (New York: Springer-Verlag, 1989). ISETL deserves a full discussion, but for now I'll just describe the process of producing ISETL/386.

```
16-bit code:
     mov es, seg _Env
     mov ax, word ptr es:_Env
     mov dx, word ptr es:_Env+2
     mov word ptr _p, ax
     mov word ptr _p+2, dx

-------------------------------

32-bit code:

     mov eax, _Env
     mov _p, eax
```

Example 1: *32- and 16-bit code generated under the large memory model*

Due to space considerations, the ISETL/386 listings are not included in this issue. They are available through *DDJ* (see the end of this article for information). The ISETL implementation consists of 29 .C files and 14 .H files, and totals about 13,000 lines of code. Some of the code is YACC output.

To sell such programs, and to acquire a program that will "bind" the DOS extender into the executable so that your customers don't need to know anything about the DOS extender, you must acquire a redistribution package from Phar Lap

When I tried to produce a 386 version of this real program, my opinion about WAT386 vs. High C nearly reversed. As long as I was working on small one- or two-module programs, WAT386's similarity to Microsoft C and Turbo C made it preferable to MetaWare High C. But once I started working on ISETL/386, with more source code, written by someone else, my allegiance shifted to High C.

High C provides better warning messages than WAT386; the High C compiler is faster than WAT386 (remember, the WAT386 compiler I used was a 16-bit real-mode program); surprisingly, High C seems to produce better overall code than WAT386; and, most important, High C and its standard library isn't buggy like WAT386.

I should mention that Watcom has terrific technical support. If you call up with a problem, you get to talk to the person responsible for the library or the compiler. Watcom is quick to find and fix bugs and, with the WPATCH utility that comes with WAT386, they have made the patch a fine art. Watcom runs a well-organized BBS. On the other hand, I don't even know how good MetaWare's technical support is, because I never needed to use it.

At one time or another, we've all thought we've found a compiler bug only to discover that in fact we have a bug in our own code. But after working with WAT386 for about a month, I found that nearly every time it was a compiler or library bug.

First of all, one of the key switch statements in ISETL was behaving bizarrely. The value of the variable being switched on was correct, we would jump to the correct case label, but a function call to *Emit(42)* wasn't working. The problem is that any constant (for example, *42*), used (anywhere) inside a switch statement is scrambled if that constant happens to match the number of case labels in the switch statement! This bug should be fixed by the time you read this. If you have this same release of the compiler, you can download a patch from the Watcom BBS.

Another problem occurs because the ISETL initialization file opens the DOS device CON (to implement a *pause( )* routine for use in ISETL programs) and tries to read from this device. The problem is, when reading from any of the DOS device files (CON, AUX, and so on), the WAT386 library gets confused between binary and text mode; a call to wait for one character actually waits for 512 characters, that makes it seem like the machine is hung.

In another project, I found that *intdosx(. . .)* was not working, even though *int386x(0x21, . . .)* worked fine. If this has not been corrected by the time you read this, a patch is available from the Watcom BBS.

In that same project, I found an obscure bug in Watcom's use of the "interrupt" keyword that had to do with calling an interrupt function rather than generating an interrupt. Basically, functions defined with *void interrupt (far *f)( )* work. But functions defined with *void (interrupt far *f)( )* (note the placement of parentheses) don't do a *PUSHFD* when you call them.

There is one problem that's not Watcom's fault: Debugging with the 386 flat memory model is hardly better than debugging in real mode. With one single segment working as a linear address space, it is nowhere as easy to catch bugs as when you have lots of little segments (for example, a 286-based protected-mode DOS extender such as DOS/16M). In fact, to debug ISETL/386, I found it necessary to cre-

(continued from page 80)
ate a DOS/16M version (ISETL/286). This shows that segmentation is not such a bad idea, after all, it's crucial for genuine memory protection. The ideal situation is to use lots of segments for development, and then switch over to the flat model for production.

The only assistance you get in catching memory protection violations from the WAT386 flat memory is the Phar Lap linker's OFFSET switch, which allows you to load code or data starting at some offset other than zero. This way, you get page faults when dereferencing bad pointers, though you often won't know where they come from.

Benchmarking with ISETL/386

Once ISETL/386 was up and running with WAT386, I was able to write some ISETL programs and use them for benchmarking the compilers. In addition to contrasting WAT386 and High C, I was able once again to compare 32-bit code with 16-bit code, using the Turbo C-produced executable from the ISETL distribution.

Figure 1 shows the results for two different ISETL programs to generate prime numbers, for an ISETL program to generate the first 1000 Fibonacci numbers, and for an overall test of ISETL operations.

	WAT386	HIGH C 386	TURBO C
PRIME.SET 2000	18.0	16.3	24.8
PRIME.SET 4000	42.6	40.0	N/A
PRIME.TUP 2000	1:03.7	52.7	1:11.3
PRIME.TUP 4000	4:14.9	3:27.6	N/A
FIB.SET 1000	15.0	14.1	20.4
FIB.SET 1200	18.0	17.0	N/A
overall test	1:05.3	59.4	1:30
total	477.5	407.1	N/A
ISETL filesize	133K	148K	209K
ISETL full compile	12:52 min.	11:45 min.	3:30 min.

Figure 1: *ISETL test execution times in seconds (Watcom and High C run times using Phar Lap 386\|DOS-Extender)*

Rather than use explicit loops, the ISETL prime number program in Listing One (page 115) uses set notation. This program creates the set of all odd numbers less than *n*, takes the union of this set with the singleton set {2}, then takes the difference between the resulting set and the set of all odd composite numbers less than *n*. The resulting set is the set of all primes <= *n*. This can be expressed in a few lines of ISETL code.

Listing Two (page 115) performs the same operation, but uses ordered tuples (sets are, of course, unordered). I had to choose a small number *n* because, even with garbage collection, ISETL gobbles up a lot of memory.

Listing Three (page 115) is a program to generate the first 1000 Fibonacci numbers. This relies on ISETL's support for assignment to the return value of a function (which allows one to write functions that "remember" past values-dynamic programming) and ISETL's arbitrary-precision arithmetic. Fibonacci(1000) is a 209-digit number. ISETL/386 takes 15 seconds to compute the first 1000 Fibonacci numbers

in the WAT386 version and 14 seconds in the High C version. The 16-bit Turbo C ISETL takes 20.4 seconds.

The High C 386 version of ISETL was faster than the WAT386 version in every case tested. Overall, the High C version was about 15 percent faster than the WAT386 version. This is somewhat surprising since, as is well known, MetaWare produces High C by using an automatic compiler-compiler (which MetaWare markets separately as the Translator Writing System).

Programs such as Phar Lap's 386|DOS-Extender and Eclipse Computer Solutions's OS/386 do not replace DOS. They manage the interface between 16-bit real-mode DOS and your 32-bit protected-mode program

Profiling with the DOS/16M protected-mode debugger from Rational Systems (DOS/16M currently has the only decent protected mode C source-level debugging tools available), I found that ISETL generally spends 50 percent of its time in only four routines. Perhaps this test is somewhat lopsided. Any real program, on the other hand, will have similar "hot spots."

The Future

Over the next few months, both Watcom and MetaWare are planning major upgrades that may be out by the time you read this. One obvious change in High C is that while the 1.5 libraries are missing functions such as *open( ), fdopen( ), dup( ), fileno( )*, and *signal( )*, High C 1.6 is scheduled to include both a Microsoft-compatible standard library (including *_dos_keep( ), int86x( )*), a 32-bit version of the GFX graphics library, and a 32-bit version of the Sterling Castle C library.

WAT386's new release should include a 32-bit protected mode source-level debugger, a 32-bit version of Watcom's graphics library (which is identical to the MSC51 graphics library), a 32-bit version of the WAT386 compiler, and a 32-bit version of Watcom's Express in-memory quick compiler. The source-level debugger is urgently needed, and should put Watcom ahead in the 386 development tool race.

A 386 compiler war may indeed be starting. While WAT386 itself is not fully mature, its arrival is a sign of the growing strength of the market for 386 development tools. And about time too, now that the first 486s are rolling off the assembly line. But remember, even an 80586 will not save you from bad code.

Product Information

Watcom C7.0/386
Watcom
415 Phillip Street
Waterloo, Ontario, Canada N2L 3X2
800-265-4555
Price: $895
Requirements: 386-based PC- or PS/2 compatible, MS-DOS 3.1 or higher, 386 DOS extender toolkit: 386|DOS-Extender (Phar Lap) or OS/386 (Eclipse Computer Solutions)

C Network Compiler/386
Novell Development Products
P.O. Box 9802
Austin, Texas 78766
512-346-8380
Price: $995

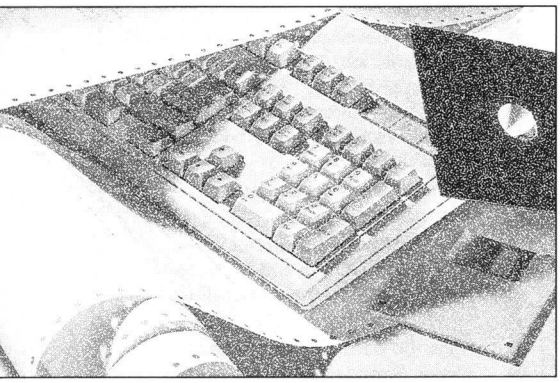
Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 115.)

Vote for your favorite feature/article.
Circle Reader Service **No. 7.**



Mixed-Language Programming with ASM

Getting the job done often requires blending models and languages

Karl Wright and Rick Schell

As applications get larger, fewer and fewer are written in a single language. Large software projects tend to come together in a piecemeal fashion — some parts are borrowed from previous projects, other parts may be purchased from various vendor sources, and, let's face it, every programmer has a favorite language. Assembly languages have made great strides recently in the area of mixed language programming. Now more than ever before, it makes sense to write applications with more than one language and to include assembly language in the mix.

Furthermore, every programming language ever created has inherent strengths and weaknesses. One area in which different languages have distinct strengths is in how procedures are called. This is an extremely important issue, because in many applications more time and effort is spent getting in and out of procedures than doing anything else! Conversely, a good choice of procedure calling conventions can actually make the difference between an application that can be written quickly and one which cannot be written at all.

Usually, higher-level languages such as C and Pascal use an argument passing technique known as the "stack frame method," where arguments are pushed onto a stack and addressed as an offset from some "frame" pointer. It is a good general technique in that it allows for an unlimited number of arguments with built-in recursion.

C and Pascal each make use of a slightly different flavor of the stack frame method. The C-style stack frame permits a variable number of arguments to be passed to a procedure. This requires that the caller remove the arguments from the stack after the procedure call, because it is the caller who knows best how many arguments were passed. In Pascal, on the other hand, the number of arguments is fixed, so the procedure itself is responsible for removing its arguments from the stack. Typically, this is done efficiently with the single machine instruction *RET xx*.

Until recently, assembly language was generally limited to what is known as the "register passing method" of passing arguments. With register passing, arguments are passed to procedures in machine registers or at fixed memory locations. (Stack frames could be constructed in assembly language, but with considerable effort on the part of the programmer.) Register passing is not a general argument passing method. There are a limited number of registers in any machine, and explicit PUSH and POP instructions must be used to retain the availability of arguments during recursion. Nevertheless, register passing is a much more efficient method of passing arguments than the stack frame when the number of arguments to a procedure is small and the particular argument registers are chosen carefully in light of the instructions, which are to be done inside the procedure.

Karl is the principal developer of Turbo Assembler and he can be reached at P.O. Box 39, Bedford, MA 01730. Rick is director of language development for Borland International and can be reached at 1800 Green Hills Road, Scotts Valley, CA 95065.

A Text "Spectrum Analyzer" Example

The example used to illustrate this point is a program that reads one or more text files, breaks them into words, and counts the individual words. It then sorts the resulting array by word count, and displays the word and the associated count together in a neat, tabular form.

This example emphasizes speed of execution, with the additional criteria that modularity is preserved and nasty tricks like self-modifying code are not used. This will permit the program to be relatively easy to change or to upgrade, and still be considerably faster than anything written wholly in a single language.

The major points that need to be covered are the interfaces between modules and what each module is responsible for, as well as the overall organization of the application.

The command line that this program will accept has the following format: *SPECTRUM <file_spec> <file_spec> . . .* where each *<file_spec>* can include wild cards. If a file name is given more than once, its spectrum will be taken more than once. The output of the application will be a table that is written to Standard Out and is sorted in order of reference count, the most referenced words being listed first.

(continued from page 84)

The basic steps are: 1. Initialize all data structures. 2. Parse the command line. For each file spec, read the file(s) and break it (them) into words. Keep a reference count for each unique word. 3. Build a list of unique words and sort it by reference count. 4. Scan the sorted list and print out the reference count and associated word for each list element.

For the sake of performance, the work of reading a file, breaking it into words, and hashing them into a symbol table is best handled in assembly language, as is the other time bottleneck that occurs when the sort is done. Less time-critical areas, such as command-line parsing and table formatting, are written in C to provide greater flexibility in the user interface. Finally, the generality of assembly language, another inherent strength, makes it best for dealing with the heap and error handling modules.

The major modules we need and their respective languages are: ERROR.ASM, the assembly language error handler (see Listing One, page 116); HEAP.ASM, the assembly language memory allocator (Listing Two, page 116); WORD.ASM, the assembly language lexer/word, table/file input (Listing Three, page 116); SORT.ASM, the assembly language general-sort procedure (Listing Four, page 119); and SPECTRUM.C, the command-line parsing, text formatting, and output written in C (Listing Five, page 120). The *make* file is shown in Listing Six, page 121.

Throughout the program, we've made every effort to use an appropriate calling convention for the situation. On procedures with stack frames, Pascal-style calling conventions are most frequently used because of their inherently faster execution and smaller code requirements. Only on procedures that require a variable number of arguments do we use a C-style stack frame.

The extensive modularity we use in this application is not absolutely necessary given its small size. We have tried, however, to put forth as general a treatment as possible, demonstrating techniques that are appropriate even for very large applications. The use of strong data abstraction is one of these techniques. In strong data abstraction, the details of an actual data structure are known only to a small set of procedures that manage that data structure. The data structure and the procedures that manage it are taken together to form a module. Any other code in the program that deals with the data structure must do so through the appropriate procedures — any other access is considered to be a breach of modularity. In this application, the HEAP and WORD modules are good examples of strong data abstractions.

The program uses SMALL model with a NEAR stack. All of the code is in segment _TEXT (except for any code in the C libraries), so CS is always set to _TEXT. Data, uninitialized data, and stacks are all in DGROUP, so SS must always be set to DGROUP. DS is also set to DGROUP in the C sections of the program, but is used as a general segment register in the assembly language code.

The interfaces to the procedures in the various modules pretty well spell out the function of each module:

Error Handling Module Because errors need only to be caught and dis-

Assembler Specific Features

The assembly language section of the application was written in Borland's Turbo Assembler 2.0 and uses several features unique to that assembler. If you are using another assembler, you may need to modify portions of the example so that your assembler will accept it. The following are the features I used and how you can work around them in your assembler.

Extended CALL automatically builds a calling stack frame by generating a series of PUSHes in the order appropriate to the specified language. For example, *CALL foo pascal,ax,bx,wordptr* would PUSH the three arguments AX, BX, and WORDPTR onto the stack in the order appropriate for Pascal stack frames, and is equivalent to

```
PUSH ax
PUSH bx
PUSH wordptr
CALL foo
```

Multiple PUSHes/POPs permit more than one item at a time to be PUSHed or POPed with a single instruction. For example,

```
PUSH AX BX
POP BX AX
```

is equivalent to

```
PUSH AX
PUSH BX
POP BX
POP AX
```

Local Symbols are enabled with the LOCALS directive. All local symbols begin with the two characters @@. They are scoped to be local to the enclosing procedure. For example

```
foo1 proc
   jmp @@exit
@@exit:  ret
endp

foo2 proc
   jmp @@exit
@@exit:  ret  ;This @@EXIT can co-
exist amicably with the former one.
endp
```

If you are using an assembler that does not support this feature, one way to work around it is to change the .MODEL statement at the start of each module to .MODEL SMALL, PASCAL. This will cause all symbols within a procedure to become local.

ARG and USES Statements the assembler used for the example has a way of setting up procedure stack frames that is somewhat easier to read than the standard method. For example:

```
foo proc pascal
arg a1,a2
uses ds,si
   . . .
```

is equivalent to the statement:

```
foo proc pascal uses ds si,a1,a2
   . . .
```

Some assemblers require a language to be specified in the .MODEL statement before the language keyword PASCAL is recognized. If this is true for your assembler, you will need to change the .MODEL statement at the start of each module to .MODEL SMALL,PASCAL.

The CODEPTR type is used occasionally in the example. It means either WORD or DWORD depending on whether the selected model has NEAR or FAR code, respectively. Because the example is SMALL model, you may replace CODEPTR with WORD wherever it is found.

— R.S.

```
void pascal ERROR_INIT (void)
        Initializes error module.

unsigned pascal ERROR_TRAP (void pascal (*execution_procedure)() )
        Returns 0 if no error occurred in the execution of
        EXECUTION_PROCEDURE or any procedures it calls. (Otherwise,
        an error code is returned.) EXECUTION_PROCEDURE is a
        generic procedure which can generate errors in its execution
        (via ERROR_LOG) and might be declared in C as follows:
        void pascal execution_procedure(void)

void pascal ERROR_LOG (unsigned error_code)
        Causes control to pass to the nearest enclosing ERROR_TRAP.
        Execution resumes with that instance of function ERROR_TRAP
        returning error_code.
```

Table 1: *Required procedures for error handling*

```
void pascal HEAP_INIT (unsigned starting_segment, unsigned segment_count)
        Initializes the heap to start at a certain segment and be
        a certain size.

void far * pascal HEAP_ALLOC (unsigned paragraph_count)
        Allocates the requested number of paragraphs from the
        heap and returns the far address of the memory in DX:AX.
        NOTE: The offset part of the address is always 0.
```

Table 2: *Required procedures for stack heap*

```
void pascal WORD_INIT (unsigned maximum_word_count)
        Initializes symbol table. The maximum number of
        different words allowed is passed so that a hash table
        can be initialized.

void pascal WORD_READ (unsigned file_handle)
        Reads all the text there is from the specified file
        handle and analyzes it.

void pascal WORD_SCAN (void pascal (*word_procedure)() )
        Calls the specified procedure once for each individual
        symbol. The word descriptor for the symbol is passed to
        WORD_PROCEDURE as an argument. WORD_PROCEDURE might
        be declared in C as follows:
        void pascal word_procedure(unsigned word_descriptor).

char far * pascal WORD_NAME (unsigned word_descriptor)
        Returns the FAR address of the name of the described symbol.

unsigned pascal WORD_REFCOUNT (unsigned word_descriptor)
        Returns the total reference count of the described symbol.

unsigned pascal WORD_COUNT (void)
        Returns the total number of distinct words processed so far.

int pascal WORD_COMPREF (unsigned word_descriptor1, unsigned
word_descriptor2)
        Compares the reference counts of two word descriptors.
        Returns flags for refcount(word_descriptor2) -
        refcount(word_descriptor1). NOTE: This procedure, while
        it obeys Pascal calling conventions, is not callable
        directly from C because it returns its result in the flag
        register. It also has the requirement that the registers
        CX and DX are preserved.

        This procedure might be described as using a sort of
        "hybrid" calling convention, where a stack frame is
        used but high-level language register conventions are not
        obeyed.
```

Table 3: *Procedures for symbol table*

(continued from page 86)

played without the ability to resume execution of the application, the error handling scheme this program uses is a mechanism whereby the stack pointer is saved at some point in the execution of the program, and if an error is encountered, the program is resumed at that point. The required procedures are listed in Table 1.

Heap Module Because data structures are allocated but never freed, a simple stack heap is the best choice for both performance and simplicity. The application uses a paragraph-based heap where memory is allocated with 16-byte granularity. This turns out to be useful because it permits any data item allocated from the heap to be described with a single 16-bit segment address. See Table 2.

Symbol Table Module The symbol table module is responsible for much of the actual work of reading in a file, converting it to words, and recording the word usage information. After it is read in, each symbol is represented by an area of memory allocated from the heap containing the reference count for the symbol and the actual text of the symbol. Because it is allocated from the heap, each symbol can be addressed by using a 16-bit word descriptor. Refer to Table 3.

Sorting Module The *sort* routine is written in assembly language because a recursive algorithm was chosen and recursion tends to be faster if register passing can be used appropriately. In this case, there are a small number of registers that are used directly; more importantly, during the innermost step of the recursion (which is done most often) no registers whatsoever need to be saved on the stack. Recursion with a stack frame can't make a decision this intelligent, because access to the arguments is needed first.

The *sort* procedure operates on an array of words, calling a generic comparison routine whose address is passed as an argument. This comparison routine uses a hybrid calling convention, where a stack frame is present but registers are not necessarily consistent with C. The level of generality this arrangement achieves is high, but it does require that the comparison routine be written in assembly language. See Table 4.

If raw speed were the only concern, the SORT_DO procedure might best be integrated entirely into the symbol table module, which would permit the comparison to be performed directly and would remove the need to call the comparison routine. But we felt that a more general treatment was superior in terms of modifiability — it is rela-

tively straightforward to add a switch to control the particular sorting method, for example.

The Command-line Parsing and Text Formatting Module We are now ready to lay out the full-scale sequencing of the program. Given the assembly language interface listed earlier, the following steps should be taken by the C portion of the program:

Assembly language's flexibility can assist in everything from optimization to the creation of programs using more than one interfacing convention

1. Allocate memory from DOS, call ERROR_INIT, and set up an error trap using ERROR_TRAP.
2. Call HEAP_INIT and WORD_INIT appropriately.
3. Parse the command line. For each file spec, call WORD_READ for all files matching the file spec (the C code is responsible for resolving all wild cards and for opening and closing each file).
4. Request the total number of unique words using WORD_COUNT, and allocate an array of 16-bit word descriptors using HEAP_ALLOC that is large enough to hold them. Call WORD_SCAN appropriately to fill up the array with word descriptors.
5. Sort the array using SORT_DO with the comparison routine WORD_COMP REF, which compares the count of references for two word descriptors.
6. Write the table title.
7. Scan the array to write out the table entries. Use WORD_REFCOUNT to get the reference count for each word descriptor, and WORD_NAME to get the name string for each word descriptor.

```
void pascal SORT_DO (unsigned far *sort_array, unsigned sort_count,
        int pascal (*compare_procedure)())
        Uses the specified compare procedure to order the array.
        COMPARE_PROCEDURE is called with two array values, and
        returns flags appropriate to a comparison of those
        values. Note that compare_procedure cannot be written in
        C because the value is returned in the machine flags. In
        addition, the segment registers are not guaranteed to be
        set up in a manner consistent with C when
        compare_procedure is called. Compare_procedure itself is
        expected to preserve CX and DX. The definition for
        compare_procedure might be stated:
        int pascal compare_procedure(unsigned value1, unsigned value2)
```

Table 4: *Procedures for sorting*

Theory of Operation

The SPECTRUM program uses a hash function and hash table to achieve its level of performance. Inside the WORD module, the procedure WORD_READ reads text into a buffer. This text is copied to a storage area one word at a time. During the copy operation, which uses the LODSB and STOSB instruc-

tions, the text is converted to uppercase and the hash value for the word is calculated, all on-the-fly.

The hash table is an array of word descriptors. An element in the hash table is 0 if there is not yet an associated symbol. The hash function is calculated by looking at each character in the word, rotating the previous hash value circularly left by five, and XORing in the character value. The final hash value is masked off to become an index into the hash table.

Now more than ever before, it makes sense to write applications with more than one language and to include assembly language in the mix

After the hash index is calculated, the corresponding hash table entry is checked. If it is 0, a new symbol is created, and its reference count is initialized to 1. Otherwise, the text of the word is compared against the text stored in the symbol whose word descriptor is found in the hash table. If it agrees, the correct symbol has been located, and its reference count is incremented. If not, a collision has occurred, and the next hash value is calculated by adding *11*2* to the current hash index (this number must be relatively prime to the size of the hash table). The process then repeats until the correct hash table entry or a 0 is found.

An unusual technique is used to speed the recognition of the various different character types during the lexing process. BX is initialized to point to a translation table, which contains a bit for each pertinent character type. An XLAT instruction followed by a *TEST AL,xxx* is then all that is needed to identify a character as a numeral, delimiter, lowercase alphabetic, and so on.

Another unusual technique is used to describe objects in the assembly language section of the program. Rather than use a full 32 bits to describe the address of a data object, which is somewhat cumbersome, a paragraph address is used instead. This paragraph address becomes the "descriptor" for the object. Data within the object is addressed by loading an appropriate segment register with the object descriptor and accessing the data with a constant offset using that segment register.

After all files have been read in and parsed, an array of word descriptors is built using the routine WORD_SCAN. This array is then sorted using SORT_DO with the comparison routine WORD_COMPREF. SORT_DO is a recursive sort that requires N*LOG(N) comparisons. It operates by dividing the array into two roughly equal parts, recursively sorting each part, and then merging the two parts in place.

Finally, to output the table, the array is scanned sequentially. For each word descriptor in the array, WORD_NAME is used to obtain the actual text of the word, and WORD_REFCOUNT is used to obtain the reference count. These values are displayed using PRINTF.

Conclusion

It is not only practical but advisable to mix languages and models in order to achieve the best results. Modern assembly language is a vital part of this mix, and will continue to be important in the future, because space and performance are always important for competitive software, no matter how powerful the hardware becomes. Assembly language's flexibility can assist in everything from optimization to the creation of programs using more than one interfacing convention.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 116.)

Vote for your favorite feature/article.
Circle Reader Service **No. 8.**

Listing One *(Text begins on page 16.)*

```
/* Sample program to copy one far string to another far string,
 * converting lowercase letters to uppercase letters in the process. */

#include <ctype.h>
char Source[] = "AbCdEfGhIjKlMnOpQrStUvWxYz0123456789!";
char Dest[100];
/* Copies one far string to another far string, converting all lower
 * case letters to upper case before storing them.  */
void CopyUppercase(char far *DestPtr, char far *SourcePtr) {
   char UpperSourceTemp;
   do {
      /* Using UpperSourceTemp avoids a second load of the far pointer
         SourcePtr as the toupper macro is expanded */
      UpperSourceTemp = *SourcePtr++;
      *DestPtr++ = toupper(UpperSourceTemp);
   } while (UpperSourceTemp);
}
main() {
   CopyUppercase((char far *)Dest,(char far *)Source);
}
```

End Listing One

Listing Two

```
; C near-callable subroutine, callable as:
;       void CopyUppercase(char far *DestPtr, char far *SourcePtr);
; Copies one far string to another, converting all lowercase letters
; to upper case before storing them. Strings must be zero-terminated.
;
parms   struc
        dw      ?       ;pushed BP
        dw      ?       ;return address
DestPtr dd      ?       ;destination string
SourcePtr dd    ?       ;source string
parms   ends
;
        .model small
        .code
        public _CopyUppercase
_CopyUppercase  proc    near
        push    bp
        mov     bp,sp                   ;set up stack frame
        push    si                      ;preserve C's register vars
        push    di
        push    ds                      ;we'll point DS to source
                                        ;segment for the duration of the loop.
        les     di,[bp+DestPtr]         ;point ES:DI to destination
        lds     si,[bp+SourcePtr]       ;point DS:SI to source
CopyAndConvertLoop:
        lodsb                           ;get next source byte
        cmp     al,'a'                  ;is it lowercase?
        jb      SaveUpper               ;no
        cmp     al,'z'                  ;is it lowercase?
        ja      SaveUpper               ;no
        and     al,not 20h              ;convert to uppercase
SaveUpper:
        stosb                           ;store the byte to the dest
        and     al,al                   ;is this the terminating 0?
        jnz     CopyAndConvertLoop      ;if not, repeat loop
        pop     ds                      ;restore caller's DS
        pop     di                      ;restore C's register vars
        pop     si
        pop     bp                      ;restore caller's stack frame
        ret
_CopyUppercase  endp
        end
```

End Listing Two

Listing Three

```
/* Sample program to copy one near string to another near string,
 * converting lower case letters to upper case letters in the process.  */

#include <ctype.h>
char Source[] = "AbCdEfGhIjKlMnOpQrStUvWxYz0123456789!";
char Dest[100];
/* Copies one near string to another near string, converting all lower
 * case letters to upper case before storing them.  */
void CopyUppercase(char *DestPtr, char *SourcePtr) {
   char UpperSourceTemp;
   do {
      /* Using UpperSourceTemp allows slightly better optimization
         than using *SourcePtr directly */
      UpperSourceTemp = *SourcePtr++;
      *DestPtr++ = toupper(UpperSourceTemp);
   } while (UpperSourceTemp);
}
main() {
   CopyUppercase(Dest,Source);
}
```

End Listing Three

Listing Four

```
; C near-callable subroutine, callable as:
;       void CopyUppercase(char *DestPtr, char *SourcePtr);
; Copies one near string to another, converting all lowercase letters to
; uppercase before storing them. Strings must be zero-terminated.
;
parms   struc
        dw      ?       ;pushed BP
        dw      ?       ;return address
DestPtr dw      ?       ;destination string
SourcePtr dw    ?       ;source string
parms   ends
;
        .model small
        .code
        public _CopyUppercase
_CopyUppercase  proc    near
        push    bp
        mov     bp,sp                   ;set up stack frame
        push    si                      ;preserve C's register vars
        push    di
        mov     di,[bp+DestPtr]         ;point DI to destination
        mov     si,[bp+SourcePtr]       ;point SI to source
        mov     cx,('a' shl 8) + 'z'    ;preload CH with lower end of
                                        ; lowercase range and CL with
                                        ; upperend of that range
        mov     bl,not 20h              ;preload BL with value used to
                                        ; convert to uppercase
CopyAndConvertLoop:
        lodsw                           ;get next two source bytes
        cmp     al,ch                   ;is the 1st byte lowercase?
        jb      SaveUpper               ;no
        cmp     al,cl                   ;is the 1st byte lowercase?
        ja      SaveUpper               ;no
        and     al,bl                   ;convert 1st byte to uppercase
SaveUpper:
        and     al,al                   ;is the 1st byte the terminating 0?
        jz      SaveLastAndDone         ;yes, save it & done
        cmp     ah,ch                   ;is the 2nd byte lowercase?
        jb      SaveUpper2              ;no
        cmp     ah,cl                   ;is the 2nd byte lowercase?
        ja      SaveUpper2              ;no
        and     ah,bl                   ;convert 2nd byte to uppercase
SaveUpper2:
        stosw                           ;store both bytes to the dest
        and     ah,ah                   ;is the 2nd byte the terminating 0?
        jnz     CopyAndConvertLoop      ;if not, repeat loop
        jmp     short Done              ;if so, we're done
SaveLastAndDone:
        stosb                           ;store the final 0 to the dest
Done:
        pop     di                      ;restore C's register vars
        pop     si
        pop     bp                      ;restore caller's stack frame
        ret
_CopyUppercase  endp
        end
```

End Listing Four

Listing Five

```
; C near-callable subroutine, callable as:
;       void CopyUppercase(char *DestPtr, char *SourcePtr);
; Copies one near string to another, converting all lowercase letters to
; upper case before storing them. Strings must be zero-terminated. Uses
; extensive optimization for enhanced performance.
;
parms   struc
        dw      ?       ;pushed BP
        dw      ?       ;return address
DestPtr dw      ?       ;destination string
SourcePtr dw    ?       ;source string
parms   ends
;
        .model small
        .data
; Table of mappings to uppercase for all 256 ASCII characters.
UppercaseConversionTable        label   byte
ASCII_VALUE=0
        rept    256
if (ASCII_VALUE lt 'a') or (ASCII_VALUE gt 'z')
      db  ASCII_VALUE           ;non-lowercase characters map to themselves

else
      db  ASCII_VALUE and not 20h  ;lowercase chars map to upper equivalents
endif
ASCII_VALUE=ASCII_VALUE+1
        endm
        .code
        public _CopyUppercase
_CopyUppercase  proc    near
        push    bp
        mov     bp,sp                   ;set up stack frame
        push    si                      ;preserve C's register vars
        push    di
        mov     di,[bp+DestPtr]         ;point DI to destination
        mov     si,[bp+SourcePtr]       ;point SI to source
        mov     bx,offset UppercaseConversionTable
                                        ;point BX to lowercase to
                                        ; uppercase mapping table
; This loop processes up to 16 bytes from the source string at a time,
; branching only every 16 bytes or after the terminating 0 is copied.
CopyAndConvertLoop:
        rept    15                      ;for up to 15 bytes in a row...
        lodsb                           ;get the next source byte
        xlat                            ;make sure it's upper case
        stosb                           ;save it to the destination
        and     al,al                   ;is this the terminating 0?
        jz      Done                    ;if so, then we're done
        endm
        lodsb                           ;get the next source byte
        xlat                            ;make sure it's upper case
        stosb                           ;save it to the destination
        and     al,al                   ;is this the terminating 0?
        jnz     CopyAndConvertLoop      ;if not, repeat loop
Done:
        pop     di                      ;restore C's register vars
        pop     si
        pop     bp                      ;restore caller's stack frame
        ret
_CopyUppercase  endp
        end
```

End Listings

Listing One *(Text begins on page 46.)*

```
;*****************************************************************************
;* File: BREAK386.ASM                                                        *
;* BREAK386 "main programs". Contains setup386, clear386, break386 and       *
;* int1_386.                                                                 *
;* Williams - June, 1989                                                     *
;* Compile with: MASM /Ml BREAK386;                                          *
;*****************************************************************************
.MODEL small
.386P

        public _break386,_clear386,_setup386,_int1_386

; Set up stack offsets for word size arguments based on the code size
; Be careful, regardless of what Microsoft's documentation says,
; you must use @CodeSize (not @codesize, etc.) when compiling with /Ml

IF @CodeSize                ; True for models with far code
arg1        EQU  <[BP+6]>
arg2        EQU  <[BP+8]>
arg3        EQU  <[BP+10]>
arg4        EQU  <[BP+12]>
ELSE
arg1        EQU  <[BP+4]>
arg2        EQU  <[BP+6]>
arg3        EQU  <[BP+8]>
arg4        EQU  <[BP+10]>
ENDIF



.DATA
; Things you may want to change:
DIRECT      EQU 0           ; IF 0 use BIOS; IF 1 use direct video access
STKWRD      EQU 32          ; # of words to dump off the stack
INTSTACK    EQU 1           ; When 0 don't display interrupt stack words
USE_INT1    EQU 1           ; Set to 0 to disable int1_386()

oldoffset   dw 0            ; old interrupt 1 vector offset
oldsegment  dw 0            ; old interrupt 1 vector segment

IF USE_INT1
video       dw 0b000H     ; segment of video adapter (changed by vinit)
csip        db 'CODE=',0
done        db 'Program terminated normally.',0
notdone     db 'Program breakpoint:',0
stkmess     db 'Stack dump:',0

vpage       db 0
vcols       db 80

IFE DIRECT
prompt      db '<V>iew output, <T>race toggle, <C>ontinue or <A>bort? ',0
savcursor   dw 0            ; inactive video cursor
ALIGN 4
vbuff       dd 1000 dup (07200720H)
ELSE
cursor      dw 0
color       db 7
ENDIF
ENDIF

.CODE

; This is the start up code. The old interrupt one vector is saved in
; oldsegment, oldoffset. int1_386 does not chain to the old vector, it
; simply replaces it.

_setup386 proc
        push bp
        mov bp,sp
        push es
        mov ax,3501H                ; get old int1 vector
        int 21h
        mov ax,es
        mov oldsegment,ax
        mov oldoffset,bx
        pop es
        mov ax,arg2                 ; get new interrupt handler address
        push ds
        mov dx,arg1
; If int1_386 is being assembled, setup386 will check to see if you are
; installing int1386. If so, it will call vinit to set up video parameters
; that int1_386 requires.
IF USE_INT1
        cmp ax,seg _int1_386
        jnz notus
        cmp dx,offset _int1_386
        jnz notus
        push dx
        push ax
        call vinit                  ; Int'l video if it is our handler
        pop ds
        pop dx
ENDIF
notus:  mov ax,2501H                ; Store interrupt address in vector table
        int 21H
        pop ds
        xor eax,eax                 ; Clear DR7/DR6 (just in case)
        mov dr7,eax
        mov dr6,eax
        pop bp
        ret
_setup386 endp


; This routine sets/clears breakpoints
; Inputs:
;    breakpoint # (1-4)
;    breakpoint type (see BREAK386.INC)
;    segment/offset of break address (or null to clear breakpoint)
; Outputs:
;    AX=0 If successful
;    AX=-1 If not successful

_break386 proc
        push bp
        mov bp,sp
        mov bx,arg1                 ; breakpoint # (1-4)
        cmp bx,1
        jb outrange
        cmp bx,4
        jna nothigh
outrange:
        mov ax,0ffffH               ; error: breakpoint # out of range
        pop bp
        ret
nothigh:
        movzx eax,word ptr arg4     ; get breakpoint address
        shl eax,4
        movzx edx,word ptr arg3     ; calculate linear address
        add eax,edx                 ; if address = 0 then
        jz resetbp                  ; turn breakpoint off!
        dec bx                      ; set correct address register
        jz bp0
        dec bx
        jz bp1
        dec bx
        jz bp2
        mov dr3,eax
        jmp short brcont
bp0:    mov dr0,eax
        jmp short brcont
bp1:    mov dr1,eax
        jmp short brcont
bp2:    mov dr2,eax
brcont:
        movzx eax,word ptr arg2     ; get type
        mov cx,arg1                 ; calculate proper position
        push cx
        dec cx
        shl cx,2
        add cx,16
        shl eax,cl                  ; rotate type
        mov edx,0fh
        shl edx,cl                  ; calculate type mask
        not edx
        pop cx
        shl cx,1                    ; calculate position of enable bit
        dec cx
        mov ebx,1
        shl ebx,cl
        or eax,ebx                  ; enable bp
        mov ebx,dr7                 ; get old DR7
        and ebx,edx                 ; mask out old type
        or ebx,eax                  ; set new type/enable bits
; Adjust enable bit (set on for data bp's, off if no data bp's)
adjge:
        mov eax,200H
        and ebx,0fffffdffH          ; reset GE bit
        test ebx,033330000H         ; test for data bp's
        jz nodatabp
        or ebx,512
nodatabp:
        mov dr7,ebx
        pop bp
        xor ax,ax
        ret
; Here we reset a breakpoint by turning off its enable bit & setting type to 0
; Clearing the type is required so that disabling all data breakpoints will
; clear the GE bit also.
resetbp:
        mov cx,bx                   ; calculate type/len bit positions
        mov edx,0fh
        dec cx
        shl cx,2
        add cx,16
        shl edx,cl
        not edx
        mov cx,bx                   ; calculate enable bit position
        shl cx,1
        dec cx
        mov eax,1
        shl eax,cl
        not ax                      ; flip bits
        mov ebx,dr7
        and ebx,eax                 ; clear enable
        and ebx,edx                 ; clear type
        jmp adjge
_break386 endp
; Reset the debug register, disabling all breakpoint. Also restore the old
; interrupt 1 vector
_clear386 proc
        pushf
        pop ax
        and ax,0FEFFH               ; turn off trace flag
        push ax
        popf
        xor eax,eax                 ; turn off all other breakpoints
        mov dr7,eax
        mov dr0,eax
        mov dr1,eax
        mov dr2,eax
        mov dr3,eax
        mov dr6,eax
        mov ax,2501H                ; restore old int 1 vector
        push ds
        mov bx,oldsegment
        mov dx,oldoffset
        mov ds,bx
        int 21H
        pop ds
```

(continued on page 98)

Listing One *(Listing continued, text begins on page 46.)*

```
        ret
_clear386 endp

IF USE_INT1
; This is all code relating to the optional INT 1 handler

; This macro is used to get a register value off the stack and display it
; R is the register name and n is the position of the register on the stack
; i.e.: outreg 'AX',10

outreg   macro r,n
         mov ax,&r
         mov dx,[ebp+&n SHL 1]
         call regout
         endm


; This is the interrupt 1 handler
_int1_386  proc far
         sti                            ; Enable interrupts (see text)
         pusha                          ; Save all Registers
         push ds
         push es
         push ss
         push @data
         pop ds                         ; Reload DS
         mov bp,sp                      ; point ebp to top of stack
IFE DIRECT
        call savevideo
ENDIF
         mov ax,video                   ; get video addressabilty
         mov es,ax
         assume cs:@code,ds:@data
         mov bx,offset notdone          ; Display breakpoint message
         call outstr
         mov edx,dr6
         call  hexout
         xor edx,edx
         mov dr6,edx
         call  crlf
;do register dump
         outreg 'AX',10
         outreg 'FL',13
         outreg 'BX',7
         outreg 'CX',9
         outreg 'DX',8
         call  crlf
         outreg 'SI',4
         outreg 'DI',3
         outreg 'SP',6
         outreg 'BP',5
         call  crlf
         outreg 'CS',12
         outreg 'IP',11
         outreg 'DS',2
         outreg 'ES',1
         outreg 'SS',0
         call  crlf
 ; do stack dump
IF STKWRD
         mov bx,offset stkmess
         call outstr                    ; Print stack dump title
         push fs
         mov dx,[ebp]                   ; get program's ss
         mov fs,dx
         mov al,'('
         call  ouch
         mov al,' '
         call  ouch
         call  hexout
         mov al,':'
         call  ouch
         mov al,' '
         call  ouch
         mov bx,[ebp+12]                ; get stack pointer (before pusha)
IFE INTSTACK
         add bx,6                       ; skip interrupt info if desired
ENDIF
          mov dx,bx
          push bx
          call  hexout
          mov al,')'
          call  ouch
          call  crlf
          pop bx
          mov cx,STKWRD
sloop:

          mov dx,fs:[bx]                ; get word at stack
          push bx
          push cx
          call  hexout                   ; display it
          pop cx
          pop bx
          inc bx
          inc bx
          loop sloop
          pop fs
ENDIF
nostack:
; Here we will dump 16 bytes starting 8 bytes prior to the instruction
; that caused the break
          push fs
          call  crlf
          mov bx, offset csip
          call outstr
          mov cx,8
          mov ax,[ebp+24]               ; get cs
          mov fs,ax
          mov bx,[ebp+22]               ; get ip
          cmp bx,8                      ; make sure we have 8 bytes before
          jnb ipbegin                   ; the begining of the segment
          mov cx,bx                     ; If not, only dump from the start
ipbegin: sub bx,cx                      ; of the segment
          push bx
          push cx
          mov dx,ax                     ; display address
          call  hexout
          mov al,':'
          call  ouch
          mov al,' '
          call  ouch
          mov dx,bx
          call  hexout
          mov al,'='
          call  ouch
          pop cx
          pop bx
          or bx,bx                      ; if starting at 0, don't display any
          jz ipskip                     ; before IP
iploop:
          mov dl,fs:[bx]                ; get byte
          push bx
          push cx
          call  hexlout                  ; output it
          pop cx
          pop bx
          inc bx
          loop iploop
ipskip:
          push bx
          mov al,'*'                    ; put '*' before IP location
          call  ouch
          mov al,' '
          call  ouch
          pop bx
; This is basically a repeat of the above loop except it dumps the 8 bytes
; starting at IP
          mov cx,8
xiploop:
          mov dl,fs:[bx]
          push bx
          push cx
          call  hexlout
          pop cx
          pop bx
          inc bx
          loop xiploop
          call  crlf
          call  crlf
          pop fs
IFE DIRECT
; Here we will ask if we should continue or abort
          mov bx,offset prompt
          call outstr
keyloop:
          xor ah,ah                     ; Get keyboard input
          int 16H
          and al,0dfh                   ; make upper case
          cmp al,'T'
          jz ttoggle
          cmp al,'A'
          jz ql
          cmp al,'C'
          jz cl
          cmp al,'V'
          jnz keyloop
; Display program's screen until any key is pressed
          call savevideo
          xor ah,ah
          int 16H
          call savevideo
          jmp keyloop

; Execution comes here to toggle trace flag and continue
ttoggle:
          xor word ptr [bp+26],256      ; toggle trace flag on stack

; Execution comes here to continue running the target program
cl:
          call  crlf
IFE DIRECT
          call savevideo
ELSE
          xor ax,ax
          mov cursor,ax
ENDIF
          pop ss
          pop es
          pop ds
          popa
; This seems complicated at first.
; You MUST insure that RF is set before continuing. If RF is not set
; you will just cause a breakpoint immediately!
; In protected mode, this is handled automatically. In real mode it
; isn't since RF is in the high 16 bits of the flags register.
; Essentially we have to convert the stack from:
;
;       16 bit Flags                32 bit flags (top word = 1 to set RF)
;       16 bit CS       to ----->   32 bit CS    (garbage in top 16 bits)
;       16 bit IP                   32 bit IP    (top word = 0)
;
; All this so we can execute an IRETD which will change RF.

          sub esp,6             ; make a double stack frame
          xchg ax,[esp+6]       ; get ip in ax
          mov [esp],ax          ; store it
          xor ax,ax
          mov [esp+2],ax        ; eip = 0000:ip
          mov ax,[esp+6]
          xchg ax,[esp+8]       ; get cs
```

(continued on page 100)

Listing One *Listing continued, text begins on page 46.)*

```
        mov [esp+4],ax
        xor ax,ax
        mov [esp+6],ax
        mov ax,[esp+8]      ; zero that stack word & restore ax
        xchg ax,[esp+10]    ; get flags
        mov [esp+8],ax
        mov ax,1            ; set RF
        xchg ax,[esp+10]
        iretd               ; DOUBLE IRET (32 bits!)

ENDIF

; Execution resumes here to abort the target program
q1:
IFE DIRECT
        call savevideo
ENDIF
        call quit
_int1_386  endp

IFE DIRECT
; save video screen & restore ours (only with BIOS please!)
; (assumes 25 lines/page)
savevideo proc near
        pusha
        push es
        mov ah,0fh
        int 10h                         ; reread video page/size in case
        mov vpage,bh                    ; program changed it
        mov vcols,ah

        push savcursor
        mov ah,3                        ; get old cursor
        mov bh,vpage
        int 10H
        mov savcursor,dx
        pop dx
        mov ah,2                        ; set new cursor
        int 10H
        movzx ax,vpage
        mov cl,vcols                    ; compute # bytes/page
        xor ch,ch
        mov dx,cx                       ; vcols * 25 * 2
        shl cx,3
        shl dx,1
        add cx,dx
        mov dx,cx
        shl cx,2
        add cx,dx
        push cx
        mul cx
        mov di,ax                       ; start at beginning of page
        pop cx
        shr cx,2                        ; # of double words to transfer
        mov ax,video
        mov es,ax
        mov si,offset vbuff             ; store inactive screen in vbuff
xloop:  mov eax,es:[di]                 ; swap screens
        xchg eax,[si]
        mov es:[di],eax
        add si,4
        add di,4
        loop xloop
        pop es
        popa
        ret
savevideo endp
ENDIF


; This routine prints a register value complete with label
; The register name is in AX and the value is in dx (see the outreg macro)
regout  proc near
        push dx
        push ax
        mov al,ah
        call  ouch
        pop ax
        call  ouch
        mov al,'='
        call  ouch
        pop dx
        call  hexout
        ret
regout  endp

; Plain vanilla hexadecimal digit output routine
hexdout proc near
        and dl,0fh
        add dl,'0'
        cmp dl,3ah
        jb ddigit
        add dl,'A'-3ah
ddigit:
        mov al,dl
        call ouch
        ret
hexdout endp

; Plain vanilla hexadecimal word output routine
hexout  proc near
        push dx
        shr dx,12
        call hexdout
        pop dx
        push dx
        shr dx,8
        call hexdout
        pop dx

; Call with this entry point to output just a byte
hexlout:
        push dx
        shr dx,4
        call hexdout
        pop dx
        call hexdout
        mov al,' '
        call ouch
        ret
hexout  endp


; These routines are for direct video output. Using them allows you to
; debug video bios calls, but prevents you from single stepping IF DIRECT
;output a character in al assumes ds=dat es=video destroys bx,ah
ouch    proc near
        mov bx,cursor
        mov ah,color
        mov es:[bx],ax
        inc bx
        inc bx
        mov cursor,bx
        ret
ouch    endp

; <CR> <LF> output.  assumes ds=dat es=video  destroys ax,cx,dx,di  clears
df
crlf    proc near
        mov ax,cursor
        mov cx,160
        xor dx,dx
        div cx
        inc ax
        mul cx
        mov cursor,ax
        mov cx,80
        mov ah,color
        mov al,' '
        mov di,cursor
        cld
        rep stosw
        ret
crlf    endp

ELSE
; These are the BIOS output routines
; Output a character
ouch    proc near
        mov ah,0eh
        mov bh,vpage
        int 10h
        ret
ouch    endp
; <CR> <LF> output.
crlf    proc near
        mov al,0dh
        call ouch
        mov al,0ah
        call ouch
        ret
crlf    endp


ENDIF

; Intialize the video routines
vinit   proc near
        mov ah,0fh
        int 10h
        mov vcols,ah
        mov vpage,bh
        cmp al,7                        ; monochrome
        mov ax,0b000H
        jz vexit
        mov ax,0b800H
vexit:  mov video,ax
        ret
vinit   endp


; outputs string pointed to by ds:bx (ds must be dat) es= video when DIRECT=1
outstr  proc near
outagn:
        mov al,[bx]
        or al,al
        jz outout
        push bx
        call ouch
        pop bx
        inc bx
        jmp outagn
outout: ret
outstr  endp


; This routine is called to return to DOS
quit    proc near
        call _clear386
        mov ax,4c00h                    ; Return to DOS
        int 21h
quit    endp


ENDIF

        end
```

End Listing One

(continued on page 100)

Listing Two *(Text begins on page 46.)*

```
;*************************************************************************
;* File: BREAK386.INC                                                    *
;* Header file to include with assembly language programs using BREAK386 *
;* Williams - June, 1989                                                 *
;*                                                                       *
;*************************************************************************
IF @CodeSize       ; If large style models
        extrn _break386:far,_clear386:far,_setup386:far,_int1_386:far
ELSE
        extrn _break386:near,_clear386:near,_setup386:near,_int1_386:far
ENDIF

; Breakpoint equates
BP_CODE     EQU  0              ; CODE BREAKPOINT
BP_DATAW1   EQU  1              ; ONE BYTE DATA WRITE BREAKPOINT
BP_DATARW1  EQU  3              ; ONE BYTE DATA R/W BREAKPOINT
BP_DATAW2   EQU  5              ; TWO BYTE DATA WRITE BREAKPOINT
BP_DATARW2  EQU  7              ; TWO BYTE DATA R/W BREAKPOINT
BP_DATAW4   EQU 13              ; FOUR BYTE DATA WRITE BREAKPOINT
BP_DATARW4  EQU 15              ; FOUR BYTE DATA R/W BREAKPOINT

; Macros to turn tracing on and off
; Note: When tracing, you will actually "see" traceoff before it turns
;       tracing off

traceon  macro
         push bp
         pushf
         mov bp,sp
         xchg ax,[bp]
         or ax,100H
         xchg ax,[bp]
         popf
         pop bp
         endm

traceoff macro
         push bp
         pushf
         mov bp,sp
         xchg ax,[bp]
         and ax,0FEFFH
         xchg ax,[bp]
         popf
         pop bp
         endm
```

End Listing Two

Listing Three

```
/*************************************************************************
* File: BREAK386.H                                                       *
*                                                                        *
* Header for C programs using BREAK386 or CBRK386                        *
* Williams - June, 1989                                                  *
*                                                                        *
*************************************************************************/

#ifndef NO_EXT_KEYS
  #define _CDECL cdecl
#else
  #define _CDECL
#endif

#ifndef BR386_HEADER
  #define BR386_HEADER

  /* declare functions */
  void  _CDECL setup386(void (_CDECL interrupt far *)());
  void  _CDECL csetup386(void (_CDECL far *)());
  void  _CDECL clear386(void);
  int   _CDECL break386(int,int, void far *);
  void  _CDECL far interrupt int1_386();

  /* breakpoint types */
  #define BP_CODE        0          /* CODE BREAKPOINT*/
  #define BP_DATAW1      1          /* ONE BYTE DATA WRITE BREAKPOINT*/
  #define BP_DATARW1     3          /* ONE BYTE DATA R/W BREAKPOINT*/
  #define BP_DATAW2      5          /* TWO BYTE DATA WRITE BREAKPOINT
  #define BP_DATARW2     7          /* TWO BYTE DATA R/W BREAKPOINT*/
  #define BP_DATAW4     13          /* FOUR BYTE DATA WRITE BREAKPOINT*/
  #define BP_DATARW4    15          /* FOUR BYTE DATA R/W BREAKPOINT*/

#endif
```

End Listing Three

Listing Four

```
;*************************************************************************
;* File: DEBUG386.ASM                                                    *
;* Example assembly language program for use with BREAK386               *
;* Williams - June, 1989                                                 *
;* Compile with: MASM /Ml DEBUG386.ASM;                                  *
;*************************************************************************
.model large
.386
INCLUDE break386.inc
.stack 0a00H

.data
align 2                                 ; make sure this is word aligned
memcell dw 0                            ; cell to write to

.code
main    proc
;setup data segment
        mov ax,@data
        mov ds,ax
        assume cs:@code,ds:@data

; start debugging
        push seg _int1_386              ; segment of interrupt handler
        push offset _int1_386           ; offset of interrupt handler
        call _setup386
        add sp,4                        ; balance stack (like a call to C)
; set up a starting breakpoint
        push seg bp1                    ; segment of breakpoint
        push offset bp1                 ; offset of breakpoint
        push BP_CODE                    ; breakpoint type
        push 1                          ; breakpoint # (1-4)
        call _break386
        add sp,8                        ; balance the stack

        push seg bp2                    ; set up breakpoint #2
        push offset bp2
        push BP_CODE
        push 2
        call _break386
        add sp,8

        push seg bp3                    ; set up breakpoint #3
        push offset bp3
        push BP_CODE
        pus' ~
        call _break386
        add sp,8

        push @data                      ; set up breakpoint #4 (data)
        push offset memcell
        push BP_DATAW2
        push 4
        call _break386
        add sp,8



bp1:
        mov cx,20                       ; loop 20 times
loop1:
        mov dl,cl                       ; print some letters
        add dl,'@'
        mov ah,2
bp2:
        int 21h
bp3:
        loop loop1                      ; repeat

        mov bx,offset memcell           ; point bx at memory cell
        mov ax,[bx]                     ; read cell (no breakpoint)
        mov [bx],ah                     ; this should cause breakpoint 4
        call _clear386                  ; shut off debugging
        mov ah,4ch
        int 21h                         ; back to DOS
main    endp
        end main
```

End Listing Four

Listing Five

```
/*************************************************************************
*                                                                        *
* File: DBG386.C                                                         *
* Example C program using BREAK386 with the built in interrupt handler   *
*                                                                        *
* Al Williams  -- 15 July 1989                                           *
* Compile with: CL DBG386.C BREAK386                                     *
*************************************************************************/
#include <stdio.h>
#include <dos.h>
#include "break386.h"

int here[10];
void far *bp;
int i;

main()
{
  int j;
  setup386(int1_386);           /* set up debugging */
  bp=(void far *)&here[2];      /* make long pointer to data word */
  break386(1,BP_DATAW2,bp);     /* set breakpoint */
  for (j=0;j<2;j++) {           /* loop twice */
    for (i=0;i<10;i++)          /* for each element in here[] */
      {
      char x;
      putchar(i+'0');           /* print index digit */
      here[i]=i;                /* assign # to array element */
      }
  break386(1,0,NULL);           /* turn off breakpoint on 2nd pass */
  }
  clear386();                   /* turn off debugging */
}
```

End Listing Five

(continued on page 104)

Listing Six *(Text begins on page 46.)*

```
;*************************************************************************
;* File: DBGOFF.ASM                                                      *
;*                                                                       *
;* Try this program if you leave a program abnormally (say, with a stack *
;* overflow). It will reset the debug register.                          *
;* Williams - June, 1989                                                 *
;* Compile with: MASM DBGOFF;                                            *
;*************************************************************************

.model small
.386P
.stack 32
.code

main proc
     xor eax,eax                        ; clear dr7
     mov dr7,eax
     mov ah,4ch                         ; exit to DOS
     int 21H
main endp
     end main
```

End Listing Six

Listing Seven

```
;*************************************************************************
;* File: CBRK386.ASM                                                     *
;* Functions to allow breakpoint handlers to be written in C.            *
;* Williams - June, 1989                                                 *
;* Compile with: MASM /Ml CBRK386.ASM;                                   *
;*************************************************************************
.MODEL small
.386P

       public _csetup386

; Set up stack offsets for word size arguments based on the code size
; Be careful, regardless of what Microsoft's documentation says,
; you must use @CodeSize (not @codesize, etc.)

IF @CodeSize               ; True for models with far code
arg1       EQU  <[BP+6]>
arg2       EQU  <[BP+8]>
arg3       EQU  <[BP+10]>
arg4       EQU  <[BP+12]>
ELSE
arg1       EQU  <[BP+4]>
arg2       EQU  <[BP+6]>
arg3       EQU  <[BP+8]>
arg4       EQU  <[BP+10]>
ENDIF



.DATA
; You may need to change the next line to expand the stack your breakpoint
; handler runs with
STACKSIZE   EQU 2048

oldoffset   dw 0                        ; old interrupt 1 vector offset
oldsegment  dw 0                        ; old interrupt 1 vector segment
oldstack    equ this dword
sp_save     dw 0
ss_save     dw 0
ds_save     dw 0
es_save     dw 0
ccall       equ this dword              ; C routine's adress is saved here
c_off       dw 0
c_seg       dw 0
oldstkhqq   dw 0                        ; Old start of stack


newsp       equ this dword              ; New stack address for C routine
            dw offset stacktop
            dw seg newstack

; Here is the new stack. DO NOT MOVE IT OUT OF DGROUP
; That is, leave it in the DATA or DATA? segment.
newstack    db STACKSIZE DUP (0)
stacktop    EQU $
            extrn STKHQQ:word           ; Microsoft heap/stack bound

.CODE
; This routine is called in place of setup386(). You pass it the address of
; a void far function that you want invoked on a breakpoint.
; It's operation is identical to setup386() except for:
;
;       1) The interrupt 1 vector is set to cint1_386() (see below)
;       2) The address passed is stored in location CCALL
;       3) DS and ES are stored in ds_save and es_save

_csetup386 proc
        push bp
        mov bp,sp
        push es
        mov ax,es
        mov es_save,ax
        mov ax,ds
        mov ds_save,ax
        mov ax,3501H
        int 21h
        mov ax,es
        mov oldsegment,ax
        mov oldoffset,bx
        pop es
        mov ax,arg2
        push ds
        mov dx,arg1
        mov c_seg,ax
        mov c_off,dx
        mov ax,seg _cint1_386
        mov ds,ax
        mov dx,offset _cint1_386
        mov ax,2501H
        int 21H
        pop ds
        xor eax,eax
        mov dr6,eax
        pop bp
        ret
_csetup386 endp

;*************************************************************************
;*                                                                       *
;* Here is the interrupt handler!!!                                      *
;* Two arguments are passed to C, a far pointer to the base of the stack *
;* frame and the complete contents of dr6 as a long unsigned int.        *
;*                                                                       *
;* The stack frame is as follows:                                        *
;*                                                                       *
;*    .                                                                  *
;*    .                                                                  *
;*   (Interrupted code's stack)                                          *
;*   FLAGS                                                               *
;*   CS                                                                  *
;*   IP <-----+                                                          *
;*   AX       |                                                          *
;*   CX       |                                                          *
;*   DX       |                                                          *
;*   BX       |                                                          *
;*   SP ------+     (Stack pointer points to IP above)                   *
;*   BP                                                                  *
;*   SI                                                                  *
;*   DI                                                                  *
;*   ES                                                                  *
;*   DS                                                                  *
;*   SS <------ pointer passed to your routine points here               *
;*                                                                       *
;* The pointer is two way. That is, you can read the values or set any of *
;* them except SS. You should, however, refrain from changing CS,IP,or SP. *
;*                                                                       *
;*************************************************************************

_cint1_386 proc
        pusha                           ; save registers
        push es
        push ds
        push ss
        mov ax,@data                    ; point at our data segment
        mov ds,ax
        mov ax,ss
        mov ss_save,ax                  ; remember old stack location
        mov sp_save,sp
        cld
        lss sp,newsp                    ; switch stacks
        mov ax,STKHQQ                   ; save old end of stack
        mov oldstkhqq,ax
        mov ax,offset newstack          ; load new end of stack
        mov STKHQQ,ax
        sti
        mov eax,dr6                     ; put DR6 on stack for C
        push eax
        push ss_save                    ; put far pointer to stack frame
        push sp_save                    ; on new stack for C
        mov ax,es_save                  ; restore es/ds from csetup386()
        mov es,ax
        mov ax,ds_save
        mov ds,ax
        call ccall                      ; call the C program
        xor eax,eax                     ; clear DR6
        mov dr6,eax
        mov ax,@data
        mov ds,ax                       ; regain access to data
        lss sp,oldstack                 ; restore old stack
        add sp,2                        ; don't pop off SS
                                        ; (in case user changed it)
        mov ax,oldstkhqq                ; restore end of stack
        mov STKHQQ,ax
        pop ds
        pop es
        popa

; This seems complicated at first.
; You MUST insure that RF is set before continuing. If RF is not set
; you will just cause a breakpoint immediately!
; In protected mode, this is handled automatically. In real mode it
; isn't since RF is in the high 16 bits of the flags register.
; Essentially we have to convert the stack from:
;
;       16 bit Flags                32 bit flags (top word = 1 to set RF)
;       16 bit CS       to ----->   32 bit CS    (garbage in top 16 bits)
;       16 bit IP                   32 bit IP    (top word = 0)
;
; All this so we can execute an IRETD which will change RF.

        sub esp,6                       ; make a double stack frame
        xchg ax,[esp+6]                 ; get ip in ax
        mov [esp],ax                    ; store it
        xor ax,ax
        mov [esp+2],ax                  ; eip = 0000:ip
        mov ax,[esp+6]
        xchg ax,[esp+8]                 ; get cs
        mov [esp+4],ax
        xor ax,ax
```

```
        mov [esp+6],ax
        mov ax,[esp+8]                  ; zero that stack word & restore ax
        xchg ax,[esp+10]                ; get flags
        mov [esp+8],ax
        mov ax,1                        ; set RF
        xchg ax,[esp+10]
        iretd                           ; DOUBLE IRET (32 bits!)
_cint1_386 endp
        end
```

End Listing Seven

Listing Eight

```
/*****************************************************************************
**
* File: CBRKDEMO.C
* Example C interrupt handler for use with CBRK386
* Williams - June, 1989
* Compile with: CL CBRKDEMO.C BREAK386 CBRK386
*****************************************************************************/


#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <dos.h>
#include "break386.h"

/* functions we will reference */
int loop();
void far broke();


main()
  {
  int i;
/* declare function broke as our interrupt handler */
  csetup386(broke);
  break386(1,BP_CODE,(void far *)loop); /* set break at function loop */

  for (i=0;i<10;i++) loop(i);
  printf("Returned to main.\n");

  clear386();                           /* turn off debugging */
  }


/* This function has a breakpoint on its entry */
loop(int j)
  {
  printf("Now in loop (%d)\n",j);
  }

/*****************************************************************************
 *
 * Here is the interrupt handler!!!
 * Note it must be a far function (normal int the LARGE, HUGE & MEDIUM
 * models). Two arguments are passed: a far pointer to the base of the stack
 * frame and the complete contents of dr6 as a long unsigned int.
 *
 * The stack frame is as follows:
 *
 *    .
 *    .
 *   (Interrupted code's stack)
 *   FLAGS
 *   CS
 *   IP <-----+
 *   AX       |
 *   CX       |
 *   DX       |
 *   BX       |
 *   SP ------+   (Stack pointer points to IP above)
 *   BP
 *   SI
 *   DI
 *   ES
 *   DS
 *   SS <------ pointer passed to your routine points here
 *
 * The pointer is two way. That is, you can read the values or set any of
 * them except SS. You should, however, refrain from changing CS,IP,or SP.
 *
*****************************************************************************/

void far broke(void far *p,long dr6)
  {
  static int breaking=1;  /* don't do anything if breaking=0 */
  int c;
  if (breaking)
    {
    int n;
    int far *ip;
/*****************************************************************************
 * Here we will read the local variable off the interrupted program's stack!
 * Assuming small model, the stack above our stack frame looks like this:
 *      i   -  variable sent to loop
 *      add -  address to return to main with
 *    <our stack frame starts here>
 *
 * This makes i the 15th word on the stack (16th on models with far code)
 *
*****************************************************************************/
#define IOFFSET 15    /* use 16 for large, medium or huge models */
    n=*((unsigned int far *)p+IOFFSET);
    printf("\nBreakpoint reached! (DR6=%lX i=%d)\n",dr6,n);
/* Ask user what to do. */
    do {
       printf("<C>ontinue, <M>odify i, <A>bort, or <N>o breakpoint? ");
       c=getche();
       putch('\r');
       putch('\n');                     /* start a new line */
       if (!c)                          /* function key pressed */
          {
          getch();
          continue;
          }
       c=toupper(c);
/* Modify loop's copy of i (doesn't change main's i) */
       if (c=='M')
          {
          int newi;
          printf("Enter new value for i: ");
          scanf("%d",&newi);
          *((unsigned int far *)p+IOFFSET)=newi;
          continue;
          }
       if (c=='A')                      /* Exiting */
         {
         clear386();                    /* ALWAYS turn off debugging!!! */
         exit(0);
         }
       if (c=='N')
         breaking=0;  /* We could have turned off breakpoints instead */
       } while (c!='A'&&c!='N'&&c!='C');
    }
  }
```

End Listings

Listing One *(Text begins on page 58.)*

```
/* segments.c */

#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include "segments.h"
#include "segtable.h"

int szAppNameLength = 8;
char *szAppName = "Segments";
char *szClocks = "Too many clocks or timers!";
char *szOutOfMemory = "Not enough memory.";

#define MAX_VARIABLE_PSEGS (MAXPSEGS - MINPSEGS - 1)

typedef struct data {
        PSEG    pseg;
        SEG             lastseg;
        SEG             oldseg;
        short   changed;
} DATA, FAR * DATAP;

PSEG    psegdata;
#define FARDATAP ( (DATAP)FARPTR(0, *psegdata) )

short   xchar;
short   ychar;
BOOL    random_action = TRUE;
int             action_count = 0;
HWND    hWindow;

PSEG allocate(LONG size, char *string);
BOOL reallocate(PSEG pseg, LONG size, char *string);
LONG FAR PASCAL SegmentsWndProc(HWND, unsigned, WORD, LONG);
int FAR PASCAL timer_routine(HWND hwnd, unsigned message, short id, LONG time);
IFP strcpyifp(IFP string1, IFP string2);
int strlenifp(IFP string);

int FAR PASCAL timer_routine(HWND hwnd, unsigned message, short id, LONG time)
{
/* Randomly allocate/free a segment in the Segment Table or
monitor the Segment Table for movement.  Update the line in the window
that changes.
*/
        int             i;
        LONG    size;
        char    buffer[40];
        RECT    rect;
        int             random_switch;
message;
id;
time;

        if (random_action)
        {
                if (++action_count < 10)
                        return(0);

                action_count = 0;

                i = rand() % MAX_VARIABLE_PSEGS;

                size = (LONG)rand();    /* 0 <= size <= 32767 */
                sprintf(buffer, "  %d bytes", (short)size);
                random_switch = rand();

                if (FARDATAP[i].pseg)
                {
                        if (random_switch > 2*32767/4)
                        {
                          if (FARDATAP[i].lastseg == 0) /* if data is free */
                              FARDATAP[i].changed = -1; /* reset the count */
                          buffer[0] = 'R';
                          reallocate(FARDATAP[i].pseg, size, buffer);
                        }
                        else if (random_switch > 1*32767/4)
                        {
                                SegmentFree(FARDATAP[i].pseg);
                                FARDATAP[i].pseg = 0;
                        }
                        else if (*FARDATAP[i].pseg)
                                DataFree(FARDATAP[i].pseg);
                }
                else
                {
                        buffer[0] = 'A';
                        FARDATAP[i].pseg = allocate(size, buffer);
                        FARDATAP[i].changed = -1;
                }

                SetRect(&rect, 9*xchar, (i+2)*ychar, 46*xchar, (i+3)*ychar);
                InvalidateRect(hwnd, &rect, TRUE);
        }

        for (i = 0; i < MAX_VARIABLE_PSEGS; i++)
        {
                if (FARDATAP[i].lastseg != *FARDATAP[i].pseg)
                {
                 FARDATAP[i].oldseg = FARDATAP[i].lastseg;
                 FARDATAP[i].lastseg = *FARDATAP[i].pseg;
                 FARDATAP[i].changed++;
                 SetRect(&rect, 9*xchar, (i+2)*ychar, 46*xchar, (i+3)*ychar);
                 InvalidateRect(hwnd, &rect, TRUE);
                }
        }
        return(0);
}

void SegmentsPaint(HDC hDC)
{
        char    buffer[100];
        short   len;
        int             i;

        TextOut(hDC, 9*xchar, ychar, "pseg  seg  oldseg moved", 23);
        for (i = 0; i < MAX_VARIABLE_PSEGS; i++)
        {
           len = sprintf(buffer, "data[%d] %.4X  %.4X", i, FARDATAP[i].pseg,

*FARDATAP[i].pseg);
                TextOut(hDC, xchar, (i+2)*ychar, buffer, len);
                    if (FARDATAP[i].pseg)
                    {
                       if (*FARDATAP[i].pseg == 0)
                       TextOut(hDC, 31*xchar, (i+2)*ychar, "Data Free", 9);
                       else
                       {
                       len = sprintf(buffer, "%.4X  %.2X", FARDATAP[i].oldseg,
                                        FARDATAP[i].changed);
                          TextOut(hDC, 21*xchar, (i+2)*ychar, buffer, len);
                                   strcpyifp(MAKEIFP(buffer, &segDgroup),
                                            MAKEIFP(0, FARDATAP[i].pseg));
                                   len = strlenifp(MAKEIFP(buffer, &segDgroup));
                          TextOut(hDC, 31*xchar, (i+2)*ychar, buffer, len);
                       }
                    }
                    else
                    TextOut(hDC, 31*xchar, (i+2)*ychar, "Free", 4);
        }

 }

IFP strcpyifp(IFP string1, IFP string2)
{
        char FAR *str1;
        char FAR *str2;

        str1 = IFP2PTR(string1);
        str2 = IFP2PTR(string2);

        while (1)
        {
                *str1++ = *str2;
                if (*str2 == 0)
                        break;
                str2++;
        }
        return(string1);
}

int strlenifp(IFP string)
{
        char FAR *str;
        int     len;

        str = IFP2PTR(string);

        for (len = 0; str[len] != 0; len++)
                ;
        return(len);
}

BOOL SegmentsInit(HANDLE hInstance)
{
    WNDCLASS   SegmentsClass;

    SegmentsClass.hCursor        = LoadCursor(NULL, IDC_ARROW);
    SegmentsClass.hIcon          = LoadIcon(hInstance,

MAKEINTRESOURCE(SEGTABLEICON));
    SegmentsClass.lpszMenuName   = "segmentsmenu";
    SegmentsClass.lpszClassName  = szAppName;
    SegmentsClass.hbrBackground  = (HBRUSH)GetStockObject(WHITE_BRUSH);
    SegmentsClass.hInstance      = hInstance;
    SegmentsClass.style          = CS_HREDRAW | CS_VREDRAW;
    SegmentsClass.lpfnWndProc    = SegmentsWndProc;

    if (!RegisterClass((LPWNDCLASS)&SegmentsClass))
        return FALSE;

    return TRUE;
}

PSEG allocate(LONG size, char *string)
{
/*
Allocate 'size' bytes from the global heap.  Copy a null terminated
'string' into the allocated memory.
*/
        PSEG    pseg;
        char FAR *farptr;
        int             i;

        if (!(pseg = SegmentAlloc(size)))
        return NULL;
        farptr = FARPTR(0, *pseg);
        for (i = 0; string[i] && i < (int)size-1; i++)
                farptr[i] = string[i];
        farptr[i] = 0;
        return pseg;
}

BOOL reallocate(PSEG pseg, LONG size, char *string)
{
/*
Allocate 'size' bytes from the global heap.  Copy a null terminated string
'string' into the allocated memory.
*/
        char FAR *farptr;
        int             i;

        if (!(SegmentRealloc(pseg, size)))
        return FALSE;
        farptr = FARPTR(0, *pseg);
        for (i = 0; string[i] && i < (int)size-1; i++)
                farptr[i] = string[i];
        farptr[i] = 0;
        return TRUE;
}
```

(continued on page 108)

Listing One *(Listing continued, text begins on page 58.)*

```
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance, LPSTR lpszCmdLine,
                int cmdShow)
{
    MSG   msg;
    HWND  hWnd;
        int   i;
        TEXTMETRIC tm;
        HDC     hdc;
        FARPROC lpprocTimer;
        DATAP   datap;
lpszCmdLine;

    if (!hPrevInstance)
        if (!SegmentsInit(hInstance))
            return FALSE;

        SegmentInit();
        if (!(psegdata =
SegmentAlloc((DWORD)sizeof(DATA)*MAX_VARIABLE_PSEGS)))
        {
                MessageBox(hWnd, szOutOfMemory, szAppName, MB_OK);
                return FALSE;
        }

        datap = FARPTR(0, *psegdata);
        for (i = 0; i < MAX_VARIABLE_PSEGS; i++)
        {
                datap[i].lastseg = 0;
                datap[i].pseg = 0;
        }

        hdc = CreateIC("DISPLAY", NULL, NULL, NULL);
        GetTextMetrics(hdc, &tm);
        xchar = tm.tmAveCharWidth;
        ychar = tm.tmHeight;
        DeleteDC(hdc);

      hWindow = hWnd = CreateWindow(szAppName, szAppName, WS_TILEDWINDOW, 0,0,
                46*xchar, 14*ychar, NULL, NULL, hInstance, NULL);

    lpprocTimer = MakeProcInstance(timer_routine, hInstance);
        while (!SetTimer(hWnd, 1, 100, lpprocTimer))
        {
                if (IDCANCEL == MessageBox(hWnd, szClocks, szAppName,
                        MB_ICONEXCLAMATION | MB_RETRYCANCEL))
        return FALSE;
        }
    ShowWindow(hWnd, cmdShow);
    UpdateWindow(hWnd);

    while (GetMessage(&msg, NULL, 0, 0))
        {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return (int)msg.wParam;
}

LONG FAR PASCAL SegmentsWndProc(HWND hWnd, unsigned message, WORD wParam,
                LONG lParam)
{
    PAINTSTRUCT ps;

    switch (message)
    {
                case WM_COMMAND:
                        switch (wParam)
                        {
                                case MENU_START:
                                        random_action = TRUE;
                                        break;
                                case MENU_STOP:
                                        random_action = FALSE;
                                        break;
                                default:
                                        break;
                        }
                    break;

            case WM_DESTROY:
                        KillTimer(hWnd, 1);
                    PostQuitMessage(0);
                    break;

            case WM_PAINT:
                    BeginPaint(hWnd, &ps);
                    SegmentsPaint(ps.hdc);
                    EndPaint(hWnd, &ps);
                    break;

            default:
                    return DefWindowProc(hWnd, message, wParam, lParam);
                    break;
    }
    return(0L);
}
```

End Listing One

Listing Two

```
/* segments.h */

#define SEGTABLEICON 1
#define MENU_START   50
#define MENU_STOP    51
```

End Listing Two

Listing Three

```
# segments.mak

cp=cl -d -DDEBUG -c -W2 -DLINT_ARGS -AM -Gswc -Os -Zdpi

.c.obj:
    $(cp) $*.c >$*.err
     type $*.err

segtable.obj: segtable.c segtable.h

segments.obj: segments.c segments.h segtable.h

segments.res: segments.rc segments.ico segments.h
    rc -r segments.rc

segments.exe: segments.obj segments.res segments.def segtable.obj
  link4 /linenumbers/co segments segtable,/align:16,/map,mlibw/noe,segments.def
  mapsym segments
  rc segments.res
```

End Listing Three

Listing Four

```
/* segments.rc */

#include "segments.h"

SEGTABLEICON  ICON    segments.ico

segmentsmenu MENU
BEGIN
    MENUITEM "Start!", MENU_START
    MENUITEM "Stop!", MENU_STOP
END
```

End Listing Four

Listing Five

```
; segments.def

NAME    Segments

DESCRIPTION 'Segments'

STUB    'WINSTUB.EXE'

CODE    MOVEABLE
DATA    MOVEABLE MULTIPLE

HEAPSIZE  10000
STACKSIZE 4096

EXPORTS
    SegmentsWndProc @1
    timer_routine @2
```

End Listings

Listing One *(Text begins on page 66.)*

```
                page    62, 132
;
;*****************************************************************************
; OBJECTS.ASM  -- This program demonstrates object-oriented programming
; techniques in 8086 assembly language.
;
dseg            segment byte public 'data'
;
; Unsigned Data Type:
Unsigned        struc
Value           dw      0
_Get_           dd      ?       ;AX = This
_Put_           dd      ?       ;This = AX
_Add_           dd      ?       ;AX = AX + This
_Sub_           dd      ?       ;AX = AX - This
_Eq_            dd      ?       ;Zero flag = AX == This
_Lt_            dd      ?       ;Zero flag = AX < This
Unsigned        ends
; UVar lets you (easily) declare an unsigned variable.
UVar            macro   var
var             Unsigned <,uGet,uPut,uAdd,uSub,uEq,uLt>
                endm
; Signed Data Type:
Signed          struc
                dw      0
                dd      ?       ;Get method
                dd      ?       ;Put method
                dd      ?       ;Add method
                dd      ?       ;Sub method
                dd      ?       ;Eq method
                dd      ?       ;Lt method
Signed          ends
; SVar lets you easily declare a signed variable.
SVar            macro   var
var             Signed  <,sGet, sPut, sAdd, sSub, sEq, sLt>
                endm
; BCD Data Type:
BCD             struc
                dw      0       ;Value
                dd      ?       ;Get method
                dd      ?       ;Put method
                dd      ?       ;Add method
                dd      ?       ;Subtract method
                dd      ?       ;Eq method
                dd      ?       ;Lt method
BCD             ends
; BCDVar lets you (easily) declare a BCD variable.
BCDVar          macro   var
var             BCD     <,bGet, bPut, bAdd, bSub, bEq, bLt>
                endm
; Declare variables of the appropriate types (For the sample pgm below):
; Also declare a set of DWORD values which point at each of the variables.
; This provides a simple mechanism for obtaining the address of an object.
                UVar    u1
U1Adr           dd      U1              ;Provide convenient address for U1.
                UVar    u2
U2Adr           dd      U2              ;Ditto for other variables.
                SVar    s1
S1Adr           dd      s1
                SVar    s2
S2Adr           dd      s2
                BCDVar  b1
B1Adr           dd      b1
                BCDVar  b2
B2Adr           dd      b2
;
; Generic Pointer Variables:
Generic1        dd      ?
Generic2        dd      ?
;
dseg            ends
;
cseg            segment byte public 'CODE'
                assume  cs:cseg, ds:dseg, es:dseg, ss:sseg
;
_This           equ     es:[bx]         ;Provide a mnemonic name for THIS.
;
; Macros to simplify calling the various methods
_Get            macro
                call    _This._Get_
                endm
_Put            macro
                call    _This._Put_
                endm
_Add            macro
                call    _This._Add_
                endm
_Sub            macro
                call    _This._Sub_
                endm
_Eq             macro
                call    _This._Eq_
                endm
_Lt             macro
                call    _This._Lt_
                endm
;*****************************************************************************
; Methods for the unsigned data type:
uGet            proc    far
                mov     ax, _This
                ret
uGet            endp
uPut            proc    far
                mov     _This,ax
                ret
uPut            endp
uAdd            proc    far
                add     ax, _This
                ret
uAdd            endp
uSub            proc    far
                sub     ax, _This
                ret
uSub            endp
uEq             proc    far
                cmp     ax, _This
                ret
uEq             endp
uLt             proc    far
                cmp     ax, _This
                jb      uIsLt
                cmp     ax, 0           ;Force Z flag to zero.
                jne     uLtRtn
                cmp     ax, 1
uLtRtn:         ret
uIsLt:          cmp     ax, ax          ;Force Z flag to one.
                ret
uLt             endp
;*****************************************************************************
; Methods for the unsigned data type.
sPut            equ     uPut            ;Same code, why duplicate it?
sGet            equ     uGet
sAdd            equ     uAdd
sSub            equ     uSub
sEq             equ     uEq
;
sLt             proc    far
                cmp     ax, _This
                jl      sIsLt
                cmp     ax, 0           ;Force Z flag to zero.
                jne     sLtRtn
                cmp     ax, 1
sLtRtn:         ret
sIsLt:          cmp     ax, ax          ;Force Z flag to one.
                ret
sLt             endp
;*****************************************************************************
; Methods for the BCD data type
bGet            equ     uGet            ;Same code, don't duplicate it.
bPut            equ     uPut
bEq             equ     uEq
bLt             equ     uLt
;
bAdd            proc    far
                add     ax, _This
                daa
                ret
bAdd            endp
bSub            proc    far
                sub     ax, _This
                das
                ret
bSub            endp
;*****************************************************************************
```

```
; Test code for this program:
TestSample      proc    near
                push    ax
                push    bx
                push    es
  Compute "Generic1 = Generic1 + Generic2;"
                les     bx, Generic1
                _Get
                les     bx, Generic2
                _Add
                les     bx, Generic1
                _Put

                pop     es
                pop     bx
                pop     ax
                ret
TestSample      endp
; Main driver program
MainPgm         proc    far
                mov     ax, dseg
                mov     ds, ax
; Initialize the objects:
; u1 = 39876.  Also initialize Generic1 to point at u1 for later use.
                les     bx, U1Adr
                mov     ax, 39876
                _Put
                mov     word ptr Generic1, bx
                mov     word ptr Generic1+2, es
; u2 = 45677.  Also point Generic2 at u2 for later use.
                les     bx, U2Adr
                mov     ax, 45677
                _Put
                mov     word ptr Generic2, bx
                mov     word ptr Generic2+2, es

; s1 = -5.
                les     bx, S1Adr
                mov     ax, -5
                _Put
; s2 = 12345.
                les     bx, S2Adr
                mov     ax, 12345
                _Put
; b1 = 2899.
                les     bx, B1Adr
                mov     ax, 2899h
                _Put
; b2 = 195.
                les     bx, B2Adr
                mov     ax, 195h
                _Put
; Call TestSample to add u1 & u2.
                call    TestSample
; Call TestSample to add s1 & s2.
                les     bx, S1Adr
                mov     word ptr Generic1, bx
                mov     word ptr Generic1+2, es
                les     bx, S2Adr
                mov     word ptr Generic2, bx
                mov     word ptr Generic2+2, es
                call    TestSample
; Call TestSample to add b1 & b2.
                les     bx, B1Adr
                mov     word ptr Generic1, bx
                mov     word ptr Generic1+2, es
                les     bx, B2Adr
                mov     word ptr Generic2, bx
                mov     word ptr Generic2+2, es
                call    TestSample

                mov     ah, 4ch                 ;Terminate process DOS cmd.
                int     21h
;
MainPgm         endp
cseg            ends
;
sseg            segment byte stack 'stack'
stk             dw      0f0h dup (?)
endstk          dw      ?
sseg            ends
;
                end     MainPgm
```

End Listing

Listing One *(Text begins on page 74.)*

```
$ PRIMES.SET
$ ISETL program to find number of primes <= n, using set notation

size := 1000 ;
sqrt_size := fix(sqrt(size)) ;
composites := {i*j : i in {3,5..sqrt_size}, j in {i..size div i}} ;
primes := {2} + {3,5..size} - composites ;
print size ;
print #primes ;
```

End Listing One

Listing Two

```
$ PRIMES.TUP
$ ISETL program to find number of primes <= n, using ordered tuples

$ tuple difference operator
diff := func(t1, t2);
    return [i : i in t1 | i notin t2 ] ;
end;

size := 1000 ;
sqrt_size := fix(sqrt(size)) ;
composites := [i*j : i in [3,5..sqrt_size], j in [i..size div i]] ;
primes := [2] + [3,5..size] .diff composites ;
print size ;
print #primes ;
```

End Listing Two

Listing Three

```
$ FIB.TUP
$ ISETL program to find Fibonacci numbers, using dynamic programming

$ uses log(): only accurate up to 308 digits
digits := func(x);
    if (x = 0) then return 1 ;
    else return 1 + floor(log(abs(x))) ;
    end;
end;

$ use "dynamic programming" to assign to fib()
fib := func(x);
    fib(x) := fib(x-1) + fib(x-2) ;
    return fib(x) ;
end;

fib(0) := 1 ;
fib(1) := 1 ;

fibonacci := [fib(x) : x in [1 .. 1000 ] ] ;
print fibonacci(1000) ;
print digits(fibonacci(1000)) ;
```

End Listings

Listing One *(Text begins on page 84.)*

```
;* Module description * This module takes care of error trapping. The scheme
;used records the trapping routine stack pointer so that an error can cause
;the stack to return to a consistent state. This module was written using
;Borland's Turbo Assembler 2.0.

;** Environment **
.model small    ;Set up for SMALL model.
locals          ;Enable local symbols.

;** Macros **
;<<Generate correct return based on model>>
procret macro
if @codesize
        retf
else
        retn
endif
endm

;** Public operations **
public pascal ERROR_INIT        ;Initialize error handler.
public pascal ERROR_TRAP        ;Set up error trap.
public pascal ERROR_LOG         ;Log error.

;** Uninitialized data **
.data?
errstk  dw ?    ;SP at last error log (-1 if none).

;** Code **
.code
;Set up DS to nothing since that is the typical arrangement.
assume ds:nothing

;[Initialize error manager]
error_init proc pascal          ;Declare proc with PASCAL calling conventions.
        mov errstk,-1
        ret
endp

;[Set up error trap]
;This procedure preserves the previous ERRSTK, sets up a new ERRSTK, and
;calls the passed procedure.  On exit, the previous ERRSTK is restored.
error_trap proc pascal          ;Pascal calling conventions.
arg @@proc:codeptr              ;Only argument is procedure to call.
uses ds,si,es,di                ;Force a save of all registers C cares for.
        push errstk
        ;Call internal routine to record return address on stack.
        call @@rtn
        pop errstk
        ret
@@rtn   label proc
        mov errstk,sp           ;Save SP so we can restore it later.
        call @@proc pascal      ;Call procedure.
        xor ax,ax               ;Return code = 0 for normal return.
        procret
endp

;[Log error]
;Control is passed to the last ERROR_TRAP, if any.
;Error code is passed and returned in AX.
error_log proc pascal
arg @@error_code:word
        cmp errstk,-1           ;Lock up if no error address.
@@1:    jz @@1
        mov ax,@@error_code
        mov sp,errstk
        procret
endp
end
```

End Listing One

Listing Two

```
;* Module description * This module manages a simple stack-based heap.
;Deallocation is not supported. NOTE: This module must be assembled with /MX
;to publish symbols in the correct case. This module is written using
;Borland's Turbo Assembler 2.0.

;** Environment **
.model small    ;Set up for SMALL model.
locals          ;Enable local symbols.

;** Equates **
err_memory = 1  ;Out of memory error number.

;** Public operations **
public pascal HEAP_INIT         ;Initialize heap.
public pascal HEAP_ALLOC        ;Allocate memory from heap.

;** External operations **

;<<Error handler>>
extrn pascal ERROR_LOG:proc     ;Long jump library procedure for errors.

;** Uninitialized data **
.data?
memptr  dw ?    ;Pointer to first free segment.
memsiz  dw ?    ;Remaining paragraphs in heap.

;** Code **
.code
;Set up DS to nothing since that is the typical arrangement.
assume ds:nothing

;[Initialize the heap]
heap_init proc pascal   ;Declare proc with PASCAL calling conventions.
arg @@start_seg:word,@@para_size:word
                        ;Arguments are starting segment and para count.
        mov ax,@@start_seg
        mov memptr,ax
        mov ax,@@para_size
        mov memsiz,ax
        ret
heap_init endp

;[Allocate memory from the heap]
heap_alloc proc pascal          ;Declare proc with PASCAL calling conventions.
arg @@para_count:word           ;Only argument is count of paragraphs.
        ;See if there is enough remaining.
        mov ax,@@para_count
        cmp memsiz,ax
        jc @@err
        sub memsiz,ax
        add ax,memptr
        xchg ax,memptr
        mov dx,ax
        xor ax,ax
        ret
@@err:  ;Out-of-memory error.
        mov ax,err_memory
        call error_log pascal,ax
        ;Never returns.
heap_alloc endp

end
```

End Listing Two

Listing Three

```
;* Module description * This module reads source files and converts them into
;words, then files the words away in a symbol table with the help of a hash
;function. This module was written using Borland's Turbo Assembler 2.0.

;** Environment **
.model small    ;Set up for SMALL model.
locals          ;Enable local symbols.

;** Equates **
;<<Error numbers>>
err_hash = 2    ;Out of hash space error number.
err_read = 3    ;Read error.

;<<Hash function>>
hash_rotate = 5 ;Amount to rotate for hash function.
hash_skip   = 11;Number of entries to skip on hash collision.

;<<Read buffer>>
rbf_size = 800h ;Size of read buffer in paragraphs.

;** Public operations **
public pascal WORD_INIT         ;Initialize hash table.
public pascal WORD_READ         ;Read file, convert to words, and hash them.
public pascal WORD_COUNT        ;Get total word count.
public pascal WORD_NAME         ;Get name of word.
public pascal WORD_REFCOUNT     ;Get reference count of word.
public pascal WORD_SCAN         ;Scan all words.
public pascal WORD_COMPREF      ;Compare word reference counts.

;** External operations **
;<<Heap>>
extrn pascal HEAP_ALLOC:proc    ;Heap allocation.

;<<Error handling>>
extrn pascal ERROR_LOG:proc     ;Trap an error.

;** Data structure **
;<<Symbol table entry>>
symtbl struc
symref  dw ?    ;Reference count.
symsiz  dw ?    ;Length of word.
ends
symnam  = size symtbl   ;Offset of start of name text.

;** Initialized data **
.data
;<<Translation character type table>>
typdlm  = 1     ;Delimiter bit.
typnum  = 2     ;Numerical digit.
typcas  = 20h   ;Lower case bit: Set if lower case letter.
xlttbl  label byte
        db '0' dup (typdlm)
        db 10 dup (typnum)
        db ('A'-1)-'9' dup (typdlm)
        db 'Z'-('A'-1) dup (0)
        db ('a'-1)-'Z' dup (typdlm)
        db 'z'-('a'-1) dup (typcas)
        db 255-'z' dup (typdlm)
;** Uninitialized data **
.data?

;<<Hash table values>>
hshptr  dw ?    ;Segment address of hash table.
hshsiz  dw ?    ;Total number of hash entries.  Must be a power of 2!
hshcnt  dw ?    ;Total free entries remaining in hash table.
hshmsk  dw ?    ;Mask for converting hash value to address.

;<<Read buffer values>>
rbfptr  dw ?    ;Segment address of read buffer.

;<<Word buffer>>
wrdbuf  db 256 dup (?)

;** Code **
.code
;Set up DS to nothing since that is the typical arrangement.
assume ds:nothing

;[Initialize hash table]
```

(Listing continued on page 118)

Listing Three *(Listing continued, text begins on page 84.)*

```
word_init proc pascal
arg @@max_word_count:word       ;Argument: Maximum number of words.
uses es,di
        ;First, allocate read buffer.
        mov ax,rbf_size
        call heap_alloc pascal,ax
        mov rbfptr,dx
        ;Now convert maximum word count to power of 2.
        mov ax,@@max_word_count
        mov cl,16+1
@@l1:   dec cl
        shl ax,1
        jnc @@l1
        mov ax,1
        shl ax,cl
        ;Initialize some hash parameters.
        mov hshsiz,ax
        mov hshcnt,ax
        dec ax
        shl ax,1
        mov hshmsk,ax
        ;Now, allocate hash table from heap.
        mov ax,hshsiz           ;Size of hash table in words.
        add ax,7
        mov cl,3
        shr ax,cl               ;Convert to paragraphs.
        call heap_alloc pascal,ax
        mov hshptr,dx
        ;Clear out hash table: 0 means 'no value'.
        mov es,dx
        xor di,di
        cld
        mov cx,hshsiz
        xor ax,ax
        rep stosw
        ret
word_init endp

;[Read file and assimilate all words]
word_read proc pascal
arg @@handle:word               ;Argument is file handle.
uses ds,si,es,di
        ;Load XLAT buffer address.  The XLAT table is used for case conversion
        ;and for character type identification.
        mov bx,offset xlttbl
@@read: ;Read next buffer while delimiter processing.
        call @@brd
        jcxz @@done
@@skip: ;Skip all delimeters, etc.
        lodsb
        xlat xlttbl
        test al,typdlm
        loopnz @@skip
        jnz @@read
        ;Adjust pointer & count.
        dec si
        inc cx
        ;If it is a number, skip to end.
        test al,typnum
        jnz @@num
        ;It is a word. We'll transfer a word at a time to the word buffer,
        ;hashing it as we go. DX will be the current hash value. CX is the
        ;amount remaining in the buffer.
        xor dx,dx
        ;Initialize output address.
        push ss
        pop es
        mov di,offset wrdbuf
@@clp:  ;Transfer.  This is THE most time-critical loop in the program.
        lodsb                   ;Read character.
        mov ah,al
        xlat xlttbl             ;Get its type.
        test al,typdlm          ;Abort if delimiter.
        jnz @@wend
        and al,typcas           ;Use case bit to convert to upper case.
        neg al
        add al,ah
        stosb                   ;Save it in word buffer.
        ;Calculate hash value.
        mov ah,cl
        mov cl,hash_rotate
        rol dx,cl
        mov cl,ah
        xor dl,al
        loop @@clp              ;Keep going until end of buffer.
        ;End of buffer while word processing.  Read more.
        call @@brd
        jcxz @@wnd2
        jmp @@clp
@@nrd:  ;Read next buffer while number processing.
        call @@brd
        jcxz @@done
@@num:  ;Numbers are not considered 'words' and should be skipped.
        ;Skip up to first delimiter.
        lodsb
        xlat xlttbl
        test al,typdlm
        loopz @@num
        jz @@nrd
        ;Adjust pointer and count.
        dec si
        inc cx
        jmp @@skip
@@done: ret
@@wend: ;End of word.  Adjust buffer pointer.
        dec si
@@wnd2: ;End of word.  Hash value is in DX, upper-case word is in WRDBUF,
        ;DI points to end of word + 1.
        push ds si cx bx        ;Save the registers we will use for this step.
        xor al,al               ;Null-terminate the word.
        stosb
        mov cx,di               ;Calculate the word's length.
        sub cx,offset wrdbuf
        mov bx,dx               ;Put the hash value in a useable register.
        shl bx,1                ;Lower bit will be discarded, so shift.
        push ss                 ;Initialize DS.
        pop ds
        assume ds:dgroup
        ;Now it is time to locate the word in the hash table if it is there,
        ;or create an entry if it is not.
@@hlp:  mov es,hshptr
        and bx,hshmsk
        mov ax,es:[bx]
        and ax,ax
        jz @@make
        ;Verify that the hash entry is the correct one.
        mov es,ax
        mov ax,cx
        cmp es:[symsiz],ax      ;Compare length of word.
        jnz @@coll
        mov si,offset wrdbuf    ;Compare actual text if that agrees.
        mov di,symnam
        repz cmpsb
        mov cx,ax
        jz @@fd
@@coll: ;Collision!  Advance to the next candidate hash entry.
        add bx,hash_skip*2
        jmp @@hlp
@@dne2: ret
@@make: ;We have encountered this word for the first time.
        ;We must create a new symbol entry of the appropriate size.
        ;First decrement remaining free hash count.
        dec hshcnt
        jz @@herr
        push cx
        push bx
        mov ax,cx               ;Calculate length of symbol descriptor.
        add ax,symnam+15
        mov cl,4
        shr ax,cl
        call heap_alloc pascal,ax
        pop bx                  ;Record symbol descriptor in hash table.
        mov es:[bx],dx
        pop cx                  ;Record length.
        mov es,dx
        mov es:[symsiz],cx
        mov di,symnam           ;Move text of word into symbol table.
        mov si,offset wrdbuf
        shr cx,1
        rep movsw
        rcl cx,1
        rep movsb
        mov es:[symref],0       ;Clear reference count.
@@fd:   ;Matching entry found!  Increment reference count.
        inc es:[symref]
@@nwd:  ;Go on to the next word in the buffer, if any.
        pop bx cx si ds
        assume ds:nothing
        jcxz @@dne2
        jmp @@skip
@@herr: ;Out of hash space error.
        mov ax,err_hash
        call error_log pascal,ax
        ;No return from ERROR_LOG.

;(Read buffer)
;Reads the next hunk of buffer.  Returns actual amount read in CX,
;DS:SI as start of data to read.
@@brd:  push dx bx
        mov cx,rbf_size*16
        mov bx,@@handle
        mov ah,3fh
        mov ds,rbfptr
        xor dx,dx
        int 21h
        jc @@err
        mov cx,ax
        xor si,si
        pop bx dx
        cld
        retn            ;Use RETN so stack frame return won't be generated.
@@err:  ;Read error.
        mov ax,err_read
        call error_log pascal,ax
        ;No return is needed because ERROR_LOG never returns.
word_read endp

;[Get total word count]
word_count proc pascal
        mov ax,hshsiz           ;Load total word capacity.
        sub ax,hshcnt           ;Subtract actual remaining free words.
        ret
word_count endp

;[Get address of name of word]
word_name proc pascal
arg @@word_desc:word            ;Argument is word descriptor.
        mov dx,@@word_desc
        mov ax,symnam
        ret
word_name endp

;[Get refcount for word]
word_refcount proc pascal
arg @@word_desc:word            ;Argument is word descriptor.
uses ds
        mov ds,@@word_desc
        mov ax,ds:[symref]
        ret
word_refcount endp
```

```
;[Scan all words]
word_scan proc pascal
arg @@scan_proc:codeptr         ;Argument is procedure to call for each word.
uses ds,si
        mov ds,hshptr
        xor si,si
        mov cx,hshsiz
        cld
@@l1:   lodsw
        and ax,ax
        jnz @@take
@@next: loop @@l1
        ret
@@take: push cx ds
        push ss
        pop ds
        call @@scan_proc pascal,ax
        pop ds cx
        cld
        jmp @@next
word_scan endp

;[Compare reference counts for two word descriptors]
word_compref proc pascal
arg @@word_desc1:word,@@word_desc2:word
uses ds
        mov ds,@@word_desc2
        mov ax,ds:[symref]
        mov ds,@@word_desc1
        sub ax,ds:[symref]
       ret
word_compref endp
end
```

End Listing Three

Listing Four

```
;* Module description * This module contains the sort routine for SPECTRUM.
;This module was written using Borland's Turbo Assembler 2.0.

;** Environment **
.model small    ;Set up for SMALL model.
locals          ;Enable local symbols.

;** Public operations **
public pascal SORT_DO           ;Perform sort.

;** Code **
.code
;Set up DS to nothing since that is the typical arrangement.
assume ds:nothing

;[Sort procedure]
sort_do proc pascal
arg @@array:dword,@@count:word,@@compare_proc:codeptr
uses ds,si,di

        ;First load up registers for internal recursion.  DS:SI will be
        ;the current sort array address, CX the count of elements to sort.
        lds si,@@array
        mov cx,@@count
        call @@sort
        ret

;Internally recursive sort routine. This routine accepts DS:SI as the sort
;array address, and CX as the count of elements to sort.
@@sort: cmp cx,2
        jnc @@go
        retn
@@go:   ;Save all registers we will change.
        ;Internally, DI and DX will be start and count of second merge area.
        push si cx di dx
        ;Divide into two parts and sort each one.
        mov dx,cx
        shr cx,1
        sub dx,cx
        call @@sort
        mov di,si
        add di,cx
        add di,cx
        xchg si,di
        xchg cx,dx
        call @@sort
        xchg cx,dx
        xchg si,di
        ;Now, merge the two areas in place.
        ;Each area must be at least size 1.
@@mrgl: ;Compare - DS:DI - DS:SI.
        call @@compare_proc pascal,ds:[di],ds:[si]
;;The following commented-out sequence is the code that would be required
;;if strict Pascal calling conventions were adhered to for calling
;;COMPARE_PROC.  You can see how much extra work this is!!
;;      push cx dx
;;      push ds
;;      mov ax,ds:[di]
;;      mov bx,ds:[si]
;;      push ss
;;      pop ds
;;      call @@compare_proc pascal,ax,bx
;;      pop ds
;;      pop dx cx
;;      and ax,ax
        jns @@ok
        ;Slide up first merge area using starting value from DI.
        mov ax,ds:[di]
```

(Listing continued on page 120)

Listing Four *(Listing continued, text begins on page 84.)*

```
        push si cx
@@sllp: xchg ax,ds:[si]
        add si,2
        loop @@sllp
        xchg ax,ds:[si]
        pop cx si
        add si,2
        add di,2
        dec dx
        jnz @@mrgl
        jmp short @@exi
@@ok:   ;Correct so far.  Advance SI.
        add si,2
        loop @@mrgl
@@exi:  ;Restore registers.
        pop dx di cx si
        retn
sort_do endp

end
```

End Listing Four

Listing Five

```
/***** File: SPECTRUM.C *****/
/* This C module is written using Borland's Turbo C 2.0 and can be
   compiled using the default switches. It should be linked with the file
   WILDARGS.OBJ from the Turbo C examples directory to enable the wild card
   file name expansion facility. Without WILDARGS, SPECTRUM will still work
   but will not be capable of expanding file names with wild cards.

   The following is an example make file, where TA is the assembler name, TCC
   is the C compiler name, TLINK is the linker name, \TC\LIB contains the C
   libraries, and \TC\EXA contains the Turbo C examples:

spectrum.exe: spectrum.obj heap.obj word.obj error.obj sort.obj
   tlink \tc\lib\c0s+\tc\exa\wildargs+spectrum+heap+word+error+sort,spectrum,,

\tc\lib\cs.lib;
heap.obj: heap.asm
        ta heap /mx;
word.obj: word.asm
        ta word /mx;
error.obj: error.asm
        ta error /mx;
sort.obj: sort.asm
        ta sort /mx;
spectrum.obj: spectrum.c
        tcc -c spectrum
*/

/*** Header Files ***/
#include <dos.h>
#include <stdio.h>
#include <fcntl.h>

/*** Function Protypes ***/
/* Used Locally */
int allocmem( unsigned, unsigned * );
int freemem ( unsigned );
int _open( const char *, int oflags );
int _close( int );
/* Error trapper */
extern void pascal error_init (void);
extern unsigned pascal error_trap (void pascal (*execution_procedure)() );
extern void pascal error_log (unsigned error_code);
/* Heap */
extern void pascal heap_init (unsigned starting_segment,
                                        unsigned segment_count);
extern void far * pascal heap_alloc (unsigned paragraph_count);
/* Symbol table */
extern void pascal word_init (unsigned maximum_word_count);
extern void pascal word_read (unsigned file_handle);
extern void pascal word_scan (void pascal (*word_procedure)() );
extern char far * pascal word_name (unsigned word_descriptor);
extern unsigned pascal word_refcount (unsigned word_descriptor);
extern unsigned pascal word_count (void);
extern int pascal word_compref (unsigned word_desc1, unsigned word_desc2);
/* Sorting procedure */
extern void pascal sort_do (unsigned far *sort_array, unsigned sort_count,
        int pascal (*compare_procedure)() );

/*** Global Variables ***/
/* Error table */
char * error_table [] = {
"Insufficient Memory\n",
"Out of Hash Space\n",
"File Read Error\n",
"Usage: SPECTRUM filespec [filespec] ... [filespec]\n(filespec may have ?,*)\n"
};

/* Arguments */
int global_argc;
char **global_argv;

/* Memory */
unsigned segment_count;
unsigned starting_segment;

/* Sort array */
unsigned sort_index;
unsigned far *sort_array;

/**** Procedures ****/
/* Fill sort array with descriptors */
```

```
void pascal array_fill(unsigned word_desc)
{
    sort_array[sort_index++] = word_desc;
}

/* Main execution procedure */
void pascal main2 (void)
{
    int i;
    unsigned j;
    int words = 0;
    int file_handle;
    if( global_argc < 2 ) {
        error_log(4);
        }
    heap_init (starting_segment, segment_count);
    word_init (32767);
    for( i=1 ; i<global_argc ; i++ ) {
        file_handle = _open (global_argv[i], O_RDONLY);
        if (file_handle != -1 ) {
            word_read( file_handle);
            _close( file_handle );
            } else {
            error_log(3);
            }
        }

    /* Obtain array address */
    sort_array = (unsigned far *)heap_alloc((word_count()+7)/8);
    /* Fill array */
    sort_index = 0;
    word_scan(array_fill);
    /* Sort array */
    printf ("Sorting...\n");
    sort_do (sort_array, sort_index, word_compref);

    /* Display output */
    printf ("\nCount\tWord\n");
    printf ("-----\t----\n");
    for (i=0 ; i<sort_index-1 ; i++) {
        j = word_refcount(sort_array[i]);
        words = words + j;
        printf ("%d",j);
        printf ("\t");
        printf ("%Fs",word_name(sort_array[i]));
        printf ("\n");
        }
    printf ("\nTotal unique words:\t%d\n",sort_index);
    printf ("Total words:\t\t%d\n",words);
}

/* Main procedure */
int main( int argc, char *argv[] )
{
    int i;
    /* Copy arguments */
    global_argc = argc;
    global_argv = argv;
    error_init();
    segment_count = allocmem(65535,&starting_segment);
    allocmem( segment_count, &starting_segment );
    i = error_trap ( main2 );
    if (i != 0) {
        /* Print error message */
        printf (error_table[i-1]);
        }
    freemem (starting_segment);
    return (i);
}
```

End Listing Five

Listing Six

```
spectrum.exe: spectrum.obj heap.obj word.obj error.obj sort.obj
  tlink /v \tc\lib\c0s+\tc\exa\wildargs+spectrum+heap+word+error+sort,

spectrum,,\tc\lib\cs.lib;
heap.obj: heap.asm
        ta heap /mx /zi
word.obj: word.asm
        ta word /mx /zi
error.obj: error.asm
        ta error /mx /zi
sort.obj: sort.asm
        ta sort /mx /zi
spectrum.obj: spectrum.c
        tcc -c -v spectrum
```

End Listings

Getting CLOS

Michael Swaine

What makes Lisp relevant today is that it is converging, in terms of features and performance, with other development environments for large software projects. When Guy Steele published *Common Lisp: The Language* (Digital Press, 1984), he codified what quickly became the de facto standard for Lisp; now the ANSI subcommittee X3J13 has nearly completed a draft standard for Common Lisp that includes the Common Lisp Object System (CLOS), an object-oriented extension to the language. I had this column half written when the second edition of Steele's book arrived, containing much new material, including an entirely new chapter on CLOS. It forced me to go back and rewrite several things; this column also corrects some things I said last month that are now out of date. Steele's treatment of CLOS is essentially the ANSI committee's treatment, and should be very close to the final draft standard, due out this year.

This convergence, though, is turning Lisp into something new. At last year's OOPSLA meeting, Bjarne Stroustrup summed up CLOS by calling it a multi-paradigm language. The circumstances (the developer of C++ being asked to deliver a lecture on the virtues of CLOS) left it unclear whether he meant it as a term of opprobrium or as a compliment.

This column's beat is paradigms, and it seemed worthwhile to take a look at how one paradigm (functional programming) is extended to another (object-oriented programming). In January we looked at "pure" Lisp; in February we saw how this pure functional paradigm has evolved with the widespread acceptance of Common Lisp, and this month we'll take a look at the objectification of Lisp in the form of the Common Lisp Object System. We'll examine two themes: How the Common Lisp data-type system underlies the CLOS class system, and how the basic concept of a function, a key aspect of Common Lisp as well as of "pure" Lisp, has been extended to the object world.

Typing Tutor

Some of the things I said last month have been superseded by the new edition of Steele's book, and this edition makes some things more official than they were previously. Because of these things and also because CLOS classes map into the Common Lisp hierarchy, I'll spell out the Common Lisp data type relationships in some detail.

To begin with, it's not really a hierarchy, but an overlapping structure that Rosemary Simpson, in her *Common Lisp: The Index* (Coral Software and Franz, Inc., 1987) calls a "heterarchy." Two types stand at the very top and bottom of the Common Lisp data type heterarchy. *t* is a supertype of every other type, and *nil* is a subtype of every other type. No object is of type *nil*. Every object is of type *t*.

The following subtypes of type *t* are of interest because X3J13 has defined them to be pairwise disjoint: *character*, *number*, *symbol*, *cons*, *array*, *random-state*, *hash-table*, *read-table*, *package*, *pathname*, and *stream*. A Common Lisp object cannot belong to more than one of these types, although it need not belong to any of them.

In addition to these types, any data type created by the *defstruct* or *defclass* macros (a user-defined structure or a CLOS class, respectively) is also disjoint from any of the above types. Any two user-defined structures are disjoint from one another unless defined otherwise, and the same goes for classes. Classes, though, are always defined in terms of other classes. I won't say much about structures here, and I'll discuss classes later.

Functions are data objects, too, and the data type *function* is disjoint from some of the above types, specifically from *character*, *number*, *symbol*, *cons*, and *array*. The types *character*, *number*, *symbol*, *cons*, *array*, and *function* are worthy of some elaboration.

Lisp Has Character

First, I'll discuss characters and numbers, correcting some outdated info from last month.

X3J13 redefined the character subtypes that were given in the first edition of Steele's book. Now the *base-character* and *extended-character* subtypes form an exhaustive partition of the type *character*. All characters are one or the other of these types. *Base-character* is implementation-defined, but must be a supertype of standard char, which is a set of 96 characters that any Lisp implementation must support; the *extended-character* type seems to be X3J13's way of dodging the confusion of bit and font attributes prevalent in Lisp.

Formerly, the data type *number* contained three disjoint subtypes, *rational*, *float*, and *complex*. Now a new type, *real*, has been introduced. The hierarchy runs like this: Types *real* and *complex* are disjoint subtypes of type *number*; other subtypes of type *number* can be defined. Each of these two subtypes also has two disjoint subtypes. Type *real* has the disjoint subtypes *rational* and *float*; it's possible to define other real subtypes. Type *rational* has the disjoint subtypes *integer* and *ratio*; other rational types can be defined.

However, type *integer* has exactly two subtypes, and Common Lisp does not allow other subtypes of *integer* to be defined. The two integer subtypes are *fixnum* and *bignum*. The *fixnum* data type is a conventional fixed-word-length integer, the word length being

implementation-dependent. *bignums* are "true" integers, their size dependent only on storage limits, not on word length. *fixnums* are more efficient than *bignums*, and are used where efficiency is more important than being able to represent precisely the number of grains of sand required to fill the universe. For example, *fixnum* is the required data type for array indices.

An object of type *ratio* represents the ratio of two integers. The Lisp system is required to reduce all ratios to the lowest terms, representing a ratio as an integer if that is possible.

Common Lisp defines four subtypes of type *float*, but an implementation need not have all four as distinct types. Types *short-float*, *single-float*, *double-float*, and *long-float*, in nondecreasing order of word length, all must be supplied, but any adjacent pair or triplet of these may be identical. Any *float* subtypes that are not identical must be disjoint.

An object of type *complex* represents a complex number in Cartesian form, as a pair of numbers. The two numbers must be of type *real*, and both must be rational or both must be of the same floating-point type.

Everything in Lisp is a List

Characters and numbers are straightforward data types, but symbols and lists are trickier. Symbols are named data objects. Type *symbol* includes among its subtypes one peculiar subtype: type *null*. *null* is the type of exactly one Lisp data object: the object *nil*. The status of type *null* is one reason that the type relationships of Common Lisp form a heterarchy rather than a hierarchy. *null* is a subtype of two types, neither of which is a subtype of the other: *symbol* and *list*. *nil* is the only object that is both a *list* and a *symbol*.

Actually, at another level, all symbols have a list-like structure. Each symbol has an associated data structure called a "property list," a list of pairs, the first elements being (typically) symbols, and the second elements being any Lisp data objects. The purpose of the property list of a symbol has evolved over time; in Common Lisp it is less important than in earlier Lisps, being used now for data not needed frequently, such as debugging, documentation, or compiler information. Neither a property list nor a symbol is of type *list*, but somehow everything in Lisp is a list of some sort. (Viewed another way, almost everything in Lisp is a function, as we'll see shortly.)

The data type *list*, though, is not regarded as being as basic as type *cons*.

These are alternate ways of viewing the same thing. A *list* is recursively defined to be either the object *nil* or a *cons* whose second component is a *list*. A *cons* is a data structure with two components, which can be pretty much anything; usually, though, the second component of a *cons* is a *list* (or *nil*, the empty list). The first components of the *cons*es making up a *list* are the elements of the list.

The data type *cons*, then, is the type of the basic data structure used to build lists. Any object that is a *cons* is also a *list*, so *list* is a supertype of *cons*. The data type *list* has exactly two subtypes, and they are disjoint: *cons* and *null*. In this sense, *null* is the (type of the) empty list. *list* itself is a subtype of the data type *sequence*, which has one other subtype: *vector*. *vector* and *list* are disjoint.

Vectors, and arrays generally, can be rather complex. Arrays can be complex, with the ability to share data with other arrays, be dynamically sized, and have fill pointers. An array that has none of these features is called a "simple array." Vectors are one-dimensional arrays; they differ from lists in performance characteristics. Accessing an element of a list is, on average, a linear function of list length, while the time to access an element of a vector is constant. When it comes to adding an element to the beginning of a list or vector, though, the relationship is reversed: constant for the list, and a linear function of vector length for the vector.

One of vector's more interesting subtypes is type *string*. Type *string* is the union of one or more vector types with the characteristic that the types of the vector's elements are subtypes of type character.

According to X3J13, the data type *function* is strictly disjoint from data types *cons* and *symbol*. But lists and symbols are the only tools available for referring to functions, or for invoking them. This is probably a use-mention distinction, but in any case, when a *list* or *symbol* is used in this way it is automatically coerced to type *function*. As we'll see shortly, there's some truth to the exaggeration that everything in Lisp is a function.

Lisp Has Class

CLOS is an object-oriented extension to CL, adding four kinds of objects to CL: classes, instances, generic functions, and methods. The key aspects are generic functions, multiple inheritance, declarative method combination, and a metaobject protocol. Classes and instances are tied to data types, generic functions to functions. I'll say only a little bit here about the metaobject protocol, which is not yet officially a part of CLOS.

The Common Lisp Object System maps classes into the data types just described. Many Common Lisp types have corresponding classes with the same names, but not all. Normally, a class has a corresponding type with the same name.

Because the types do not form a simple tree, and a type can be a subtype of two types neither of which is a subtype of the other, you might expect CLOS to support multiple inheritance, in which a class can inherit from more than one superclass. In fact, this is the case. The heterarchical structure of types is mirrored in the inheritance structure of classes, but CLOS requires that more structure be added to establish a clear precedence order for inheritance. For example, the class *vector* has superclasses *sequence* and *array*, just as the type *vector* has supertypes *sequence* and *array*, but from which superclass does *vector* inherit what?

CLOS resolves questions such as this by requiring that you specify an ordering of direct superclasses when you define a class (and by supplying this ordering for predefined classes). The business of deriving a full precedence order is fairly complex, but the CLOS class precedence order for predefined classes resolves such issues. In particular, the precedence order for the class *null* is *null*, *symbol*, *list*, *sequence*, *t*; and the precedence order for the class *string* is *string*, *vector*, *array*, *sequence*, *t*. By implication, the precedence order for the class vector is *vector*, *array*, *sequence*, *t*; so array methods have precedence over sequence methods when class vector is inheriting methods.

Everything in Lisp is a Function

The simplifying generalization is that everything in Lisp is a function. It's nearly true; any data object can be treated as a function, or rather, as a form. A form is simply a data object treated as a function. You treat a data object as a function when you hand it to the evaluator, which is the mechanism that executes Lisp programs. The evaluator accepts a form and does whatever computation the form specifies.

The evaluator can be implemented in various ways, such as by an interpreter that traverses the form recursively, performing the required calculations along the way; or as a pure compiler; or by some mixed form. Common Lisp requires that correct programs produce the same results, regardless of the method of implementation. The evaluator is available to the user via the function *eval*, and also the special form *eval-when*, which allows specifying that a form should be evaluated, say, only at compile time.

Not every data object specifies a meaningful function, but most do. To the evaluator, there are three kinds of forms, corresponding to three nearly disjoint data types. There are symbols, lists, and self-evaluating forms (per X3J13, all standard Common Lisp objects, except symbols and lists, are self-evaluating forms).

Self-evaluating forms are taken literally by the evaluator; they return themselves on evaluation.

Symbols name variables, constants, keywords and functions. They evaluate to whatever they name; for example, what they are bound to or what they are set to.

Lists, from the viewpoint of the evaluator, come in three varieties: special forms, macro calls, and function calls. Note that while a function is not a list, a function call is.

Special forms are structural elements of the language that don't fit the functional paradigm well, such as the if-then-else structure. These deviations from the purity of the paradigm have been a part of Lisp since the beginning, and new special forms have been added over the years, but in Common Lisp the set of special forms is fixed and cannot be extended by the programmer. A macro is a function from forms to forms, much as in other languages. A macro call, when evaluated, is said to be expanded. Programmers can extend the set of macros. Despite the fact that they are not true functions, special forms look like functions syntactically, as do

macros. The consequence of this is that when you are sitting at the keyboard typing in Lisp code, it feels like you are dealing with one kind of construct: A parenthesized list that represents a function and its arguments.

A form that is a function call consists of a list whose first element is a function name. The other elements of the list, if any, are treated by the evaluator as forms to be evaluated to provide the function with arguments. There are two levels of evaluation that take place whenever the evaluator deals with a function call: The arguments get evaluated, then the function is evaluated with these arguments. Typically, the evaluation of the function produces a value, which becomes the value of the original form.

There are two ways in which the first element of a form can name a function, one involving a symbol and the other involving a list. Because symbols are used to name functions, this is the most direct and obvious way. The other way involves the use of a lambda expression. A lambda expression is technically not a form, and cannot be evaluated. It is a list, the first element being the word lambda. The second element is a list of parameters, and this is followed by some number of forms to be evaluated, which can use the parameters. When the function that the lambda expression names is applied to arguments, the parameters are bound to the arguments and the forms are executed with these bindings.

Using a lambda expression as a function name is like slipping physical actions into your speech, as you would be doing if you referred to what comes at the end of a joke by making a punching motion, then saying the word "line." Lambda expressions see their main use in defining functions, roughly like this:

```
defun <fn-name> <lambda-list>
                        <forms>
```

CLOS adds generic functions to Lisp. Because the evaluation of functions is central to Lisp, the extension of functions to generic functions has a lot to say about how it feels to program in CLOS.

A generic function is a true Lisp function, is called with the same syntax, and can be used in the same contexts in which a Lisp function can be used.

Defining a generic function object is similar to defining a function. You use the *defgeneric* macro, basically like this:

```
defgeneric <fn-name> <lambda-list>
                        <methods>
```

The difference is that, rather than a fixed set of forms to be evaluated, the generic function has a collection of method descriptions, each of which may consist of a number of forms. The method descriptions have their own lambda lists that must be congruent with the main lambda list. Texas Instruments has implemented generic functions in its TICLOS as normal compiled functions with pointers to data structures containing their slots. When the function is called, it is up to the object system to select the appropriate method from its methods. Actually, not select; the technique is more general than this, and is called "method combination." The code eventually executed is called the "effective method."

The selection/combination has three stages: select applicable methods, order them by precedence, and apply method combination. The method combination, defined in the definition of the generic function, can be as simple as using the most specific method, or it can be some function of some of the applicable methods. Some built-in method combination types are +, *and*, *or*, *append*, *max*, and *min*, which perform the corresponding functions on the applicable methods to produce the effective method.

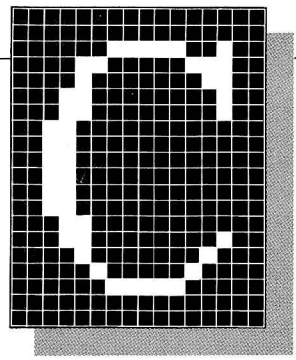
Some of the most interesting CLOS functions are those that allow customization of the object system itself, by manipulating metaobjects and metaclasses. Unfortunately, these have not yet been approved by X3J13 for inclusion in the standard. They do, however, support the original spirit of Lisp as an introspective language, with all the strangeness that Douglas Hofstadter suggested when I quoted him last month, a quote that I here double-quote:

"A . . . double-entendre can happen with LISP programs that are designed to reach in and change their own structure. If you look at them on the LISP level, you will say that they change themselves; but if you shift levels, and think of LISP programs as data to the LISP interpreter . . . then in fact the sole program that is running is the interpreter, and the changes being made are merely changes in pieces of data."

Editor's Note: *For a general discussion of functional programming, see "Functional Programming and FPCA '89" by Ronald Fischer,* DDJ, *December 1989. Also, see "A Neural Networks Instantiation Environment" by Andrew J. Czuchry, Jr. in next month's* DDJ *for more information on programming in Lisp.*

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 10.**



A Thousand CURSES on TEXTSRCH

Al Stevens

Last month we completed the retrieval processes of the TEXTSRCH project, a "C Programming" column project that we started in December of last year. It builds and maintains a text indexing and retrieval data-base system that allows a user to find text files by composing key word query expressions. The program has two passes: an index builder and a query retrieval program. The query retrieval program searches the text file indexes for files that match the criteria of a Boolean key word search. It delivers a list of the file names that match the search. With the software, developed through last month's installment, a user can determine which files in the text data base match the criteria of the query, and from there he or she can move the files into another application, for example, a word processor.

This month we will add a new feature to TEXTSRCH to allow the user to select and view one of the files from within the TEXTSRCH retrieval program itself. Instead of merely displaying a list of file names that match the query, TEXTSRCH will display them in a menu window from which the user can select. Then it will display the contents of the selected file with the query expression's key words highlighted.

We use this new feature to explore the screen driver software called "CURSES." CURSES is a library of functions that were originally implemented in Unix V. Its purpose is to allow you to write portable, terminal device-independent C programs. The Unix system and the C language are still inexorably oriented to the simple teletype-like console device. The standard input and output devices are such that they can be anything from a clunky old ASR-33 teletype to a high-resolution, many MIPS, full-color, belch-fire, neck-snapper graphics workstation. To support them all, *stdin* and *stdout* must speak to the lowest-common denominator.

There are still many installations that use simple terminal devices, and these devices are grist for the *stdin, stdout* mill. Terminals are the same yet they are different. A system's local devices may be many and varied, and the remote dial-up users are likely to be calling in from any one of a number of different terminal types. These different video display terminal devices can work as one because they share the common ability to send and receive ASCII text with carriage returns and line feeds. If that is the only way a program needs to communicate with a user, then these devices share all the commonality they will ever need.

There are, however, features in the typical video display terminal that a program can use to enhance its user interface. Most such terminals have command sequences to clear the screen, position the cursor, and so forth. As you might expect, there is no one way to do all this. ANSI published a standard, and some terminal devices comply. The ANSI.SYS device driver that comes with MS-DOS allows a PC to use the ANSI protocols.

Many terminals have their own, non-ANSI ways to clear the screen, position the cursor, scroll, and achieve other video effects. A program written specifically to use the features of one of these terminals must be modified if an incompatible terminal is connected to the program. As a programmer in such an environment you have three choices: You can write to the common base, which means simple, unadorned, glass-teletype ASCII text; you can use the unique features of the terminal du jour and modify your program every time a new terminal comes into the picture; or you can write to a higher-level video protocol and have a system-level interpreter library translate your video commands into the commands of whatever terminal a user signs on with. The first choice is the appropriate one for text filter programs and console command programs. The second one is appropriate when the operating environment is well-defined and contained, and perhaps when user language performance is an issue. The third choice is the best one to make when you are striving for portability and device independence.

CURSES

To provide for an environment where users with different terminals can use the same software, and where the software can use the video terminal features that go beyond simple text display, the Unix system contains the "CURSES" library and the "termcaps" data base. The data base describes the video protocols of each of the terminals, and the library provides functions that translate a higher-level common protocol into that of the user's terminal device.

CURSES functions facilitate a primitive window-oriented display architecture. You can define windows and use them as virtual terminals. There are character and string display operations, cursor positioning operations, video attributes (such as highlighting and normal displays), keyboard character and string input, scrolling, and simple text editing operations such as inserting and deleting characters and lines.

CURSES works in memory buffers. You address your operations to a defined window, and CURSES makes the changes in memory. These changes do not appear on the screen until you tell CURSES to refresh the window. This method might seem peculiar to a PC

programmer who is accustomed to instantaneous video memory updates. But it reflects its roots in the RS-232 ASCII terminal. It takes more time to update a terminal's screen than it does to write characters into a PC's video memory. For example, a 24 × 80 terminal operating at 19,200 baud will use about a second to refresh its screen. A well-behaved video library can keep a copy of the current screen image and be building another copy to contain whatever changes you are making. When you tell it to refresh, the library can, if the terminals features allow, refresh only that part of the screen that changes.

Lattice C 6.0

Lattice C is an old PC workhorse that has been around since Gates was in short pants and Kahn was a dynasty. It was one of the original full K&R C compilers for the PC. The first Microsoft C was in fact Lattice in a Microsoft binder giving Microsoft an entrance into the C compiler marketplace while they took their time building one of their own. Because Microsoft's own C compiler targeted upward compatibility for programs written with their earlier Lattice version and because the rest of the C compiler business strives for compatibility with Microsoft C, it can be said that Lattice had a strong influence on what C compilers for the PC would become.

There are Lattice versions now for other platforms, including the amazing and wonderful Commodore Amiga. The most recent version for the PC, Version 6.0, supports DOS and OS/2, conforms with the ANSI proposed draft, and comes with a source-level debugger, an editor, an assembler, a librarian, a linker, lots of utility programs, a communications function library, a database library that supports dBase III formats, a graphics library, a library of DOS-OS/2 Family Mode functions, and a CURSES library. If you do not require an Integrated Development Environment after the fashion of Turbo C, QuickC, and others (and many of us do not), this is as complete a C language development environment as you'd want.

The Lattice CURSES Library

The Lattice CURSES library is available in source code for $125 so you can port it to the compiler of your choice. This CURSES library provides a means for developing screen programs that can be ported between DOS, OS/2, and Unix with minimum changes. I used the Lattice compiler and this library to build the document viewing feature that we are adding to TEXTSRCH this month.

Porting Crotchety TEXTSRCH to Lattice C

I wrote the first three installments of TEXTSRCH in Turbo C 2.0. My intention was to make the code as close to ANSI C and as far from the PC architecture as possible to avoid restricting the program to a particular platform. To use the Lattice CURSES library, I decided to port the code to Lattice C rather than to port the CURSES code to Turbo C. Somehow I figured I'd have an easier time of it by porting my own stuff. Maybe, maybe not.

CURSES is a library of functions that were originally implemented in Unix V to allow you to write portable, terminal device-independent C programs

The port was reasonably easy with just a few hitches. Here is what I ran up against, and what follows is a new crotchet that I hereby induct into the "C Programming" column Crotchet Hall of Fame.

It is said that a compiler that compiles programs that comply with the ANSI standard is considered to be an ANSI-conforming compiler. But what about those compilers that extend the standard? For example, Watcom C supports the C++ convention for double-slash comments. The Turbo C *fopen* function allows the use of a non-standard mode parameter. To be sure, both compilers will compile programs that do not use these extensions. But, because you can write programs that use them, you can unintentionally write code that is not ANSI-conforming. Turbo C has, of course, many other extensions, such as pseudo-register variables and interrupt functions. Many compilers now include the interrupt function type, which I first saw in Wizard C, the ancestor of Turbo C. Usually you can tell the compilers to disallow such extensions, that you are interested in writing portable code, and the compilers will comply. But when an extension takes the form of the values accepted as a function's parameters, the compiler does not preempt the extension. So, in all my innocence and with good intentions aforethought, I used the Turbo C "rt" and "wt" formats for the *fopen* mode parameter. The Lattice *fopen* function, in true ANSI compliance, simply refused to open those files because it did not recognize the modes. Turbo C also supports mode formats such as "r+b" where ANSI and the Lattice documentation specify "rb+." Naturally, I used the non-standard formats in my *fopen* calls. You should go through all the code in <index.c> from last month and change every "r+b" to "rb+" and "w+b" to "wb+." Change all *fopen* modes that include the "t" to remove the "t." I believe that the definition of compliance should exclude such extensions.

The next portability issue came with header files. Turbo C puts some function prototypes into more than one header file. In this case, I included the non-standard <process.h> to get the prototype for the *exit* function. According to ANSI, this prototype is in <stdlib.h>, and that is where Lattice keeps it.

The moral of the story has to be: Get a good ANSI function library reference book and ignore the library documentation that comes with your compiler.

Other storms in my port were the result of issues unrelated to ANSI C. The TEXTSRCH <cmdline.c> source file uses the Turbo C *findfirst* and *findnext* functions to search a file directory. ANSI C has no equivalent functions because, I suppose, there are some C platforms that have no analogue to the DOS directory search. When I wrote about those functions last month, I said you would need to make substitutions if you are using a different compiler. Now I find myself in that same boat. Because Lattice has equivalent functions in its *dfind* and *dnext* functions and because it does not have the <dir.h> file that cmdline.c includes, I coded a <dir.h> that substitutes with macros the Lattice functions for the Turbo C functions. You will find <dir.h> as Listing One on page 144.

I had the global variable OK defined as 0, and the Lattice <curses.h> defines it as 1. If you use the Lattice definition, all the TEXTSRCH code works fine.

The next set of problems occurs because of errors in the Lattice header files. It's difficult to imagine how these errors have gone undetected until now. The <curses.h> file includes definitions of keystroke values for the keypad keys. One of these is *KEY_PGDN*, which defines the value returned when you press the PgDn key. The definition, 0x0181, is wrong. It should be 0x0151. The

(continued from page 128)
macros for the CURSES *wstandout* and *wstandend* functions are incorrect. They do not include the *win* parameter in the macro expansion. Not only do you get compiler warnings, but the functions do not work. Finally, the Lattice <stdlib.h> header file specifies in the *free* function prototype that *free* returns *void*, which is wrong. It returns *int*. I had to repair the Lattice header files to proceed.

My final problem was with the CURSES screen driver software. For some reason it reprograms the video mode of my Vega Video 7 in a way that makes the display go off into the weeds at unexpected times, usually after I exit my program. To solve this problem I would need to look at the source code for CURSES, and time and deadlines do not permit. A workaround solution is to run the TEXTSRCH program from a batch file that executes the DOS command MODE CO80 after the TEXTSRCH program exits to DOS.

TEXTSRCH

To install the new file-viewing functions of TEXTSRCH, you must replace the source file named <search.c> from last month with the one in Listing Two, page 144. You must also compile and link <display.c>, Listing Three, page 144, and <error.c>, Listing Four, page 149, into the <textsrch.exe> program.

The BLDINDEX program works the same way that it did before. The new feature is in the TEXTSRCH program. When you enter a query expression the results are now displayed in a screen window with an ASCII -> cursor to the left of each file name. With the up and down arrow keys, you move that cursor and scroll the display. When the cursor points to a file you might want to view, press the Enter key. The first page of the selected document text displays in a new full-screen window. The up and down arrow keys will scroll the display. The up and down page keys will page the display. The Home key goes to the first page and the End key to the last. During the display all occurrences of the key words from the query expression display in a highlighted mode. You can move to the next page where a key word appears by pressing the right arrow key. The left arrow key moves you to the previous page where a key word appears.

Here is how to use CURSES to achieve these results. The *process_result* function in <search.c> is changed. Instead of displaying the matching file names it builds an array of those names. Then it calls the CURSES *initscr* function to initialize the screen manager, calls *select_text* so the user can select a file to look at, and calls the CURSES *endwin* function to shut down the screen manager.

The *select_text* function is where the user picks a file to view. We use the CURSES *newwin* function to build a menu window. The *keypad* function allows the CURSES keyboard routines to recognize the keypad characters, and the *wsetscrreg* function defines the scrolling boundaries of the window. Use of this function prevents the window borders from scrolling along with the rest of it.

The *display_page* function displays a specified page of the file menu in the window. Initially we call it to display the first page. Then we draw a box around the window, write the ASCII -> selector cursor, and read the keyboard. The various cases under the keystroke switch, take care of moving the selector cursor up and down, and paging and scrolling the file selector menu. When the user presses the Enter key, that case calls the *display_text* function, passing the name of the selected file as shown in the menu window.

At this point we must consider the values assigned to the different keys we are interpreting. They are taken

(continued from page 130)
from the Lattice <curses.h> header file and they correspond to what the Lattice version of the CURSES *wgetch* function returns for the cursor keys when the CURSES *keypad* mode is on. These values might not apply to different environments. Also see the use of the *VERT_DOUBLE* and *HORIZ_DOUBLE* global variables in the call to the CURSES *box* function. These too appear in <curses.h> and they correspond to the PC's graphics characters for border characters. You might need to change these values to something that matches your system. CURSES does not provide for border corner characters, but the Lattice implementation recognizes the IBM set and uses the matching corner graphics characters.

Look now at Listing Three to see the code that displays a text file. The function named *display_text* opens the file and calls its *do_display* function if the file opens OK. If not, it calls the *error_handler* function that you will find in Listing Four. This general-purpose function displays an error message in a window, waits for a key press, and clears the message.

The *do_display* function reads all the lines of text from the chosen file and stores them in a linked list in the heap. The list connects each line to its following line and records the positions of any key words in each line.

The *findkeys* function takes care of finding and storing key word occurrences. It scans the line of text comparing each word to the ones in the query expression. If a word matches one of the keys, its character offset relative to the start of the line goes into the header block of the line's linked list entry. The header block can contain up to five key words for each line, which should be enough to call your attention to the line.

After all the lines of text are tucked away in the linked list, the program builds a full-screen window to display the text. The *display_textpage* function displays a page of text beginning with a specified line. It displays the lines a character at a time. If the current character position is marked in the line's header block as the position of a key word, the program calls the CURSES *wpstandout* function to cause the word to be highlighted. When the program finds the next white space character, it calls the CURSES *wpstandend* function to return the display to the normal, non-highlighted mode.

Once a page is displayed, the program reads the keyboard. As with the file selector menu, the keystroke values control the screen display. You can

page and scroll up and down, and you can move the next or previous page where a marked key word appears. The *pagemarked* function makes this test, finding the first line of the specified page and looking at each entry in the list to see if any line has a marked key word.

When you press the ESC key, the function calls *wclear* to clear the text display window and *wrefresh* to refresh that clearing to the screen. Then it deletes the window and frees the heap of the linked list entries.

Back in the *select_text* function the file selector window gets redisplayed and the user can pick out another file to look at.

TEXTSRCH Performance

How effective is the CURSES approach to the development of portable code? The proof would be in the successful porting of a program such as this one to another platform. I am sure that this program would port to a Unix system with no more fuss than I had moving it to Lattice C. There is, however, one big area of concern in such a move. We do not know how efficiently the program would operate. CURSES is a technique for the portability of screen driver code to a multitude of display devices. Its implementation in the Lattice library makes for an effective and efficient program because they used all the PC tricks for fast screen updates. What's more, I developed this program on and for a 20-MHz 386 computer. The only way to know how well or poorly this particular use of CURSES would work on a slower machine or with a different terminal is to move the program. So, with that in mind, I moved TEXTSRCH to the slowest compatible computer at my house, an 8-MHz COMPAQ II. I am happy to report that it works fine. This does not, however, qualify it for an environment where the terminal device is a serial VDT. I would suspect that some of the ways I used CURSES are not the best choices for such a setup. A seasoned CURSES programmer probably knows intuitively what to do and what to avoid to support the most effective user interface.

The collective abilities and shortcomings of CURSES across a wide selection of terminals would, no doubt, influence the way you would design a user interface. Given that one could learn these boundaries and with all this in mind, I can conclude that CURSES is an effective technique for wide platform independence of text-based screen management. That, of course, is no news to Unix programmers, who have had CURSES for several years. It is news to those others of us who might be looking for tidy ways to develop programs on the PC that can be moved to other operating environments.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 144.)

Vote for your favorite feature/article.
Circle Reader Service **No. 11.**

Sifting for Sharks' Teeth

Prowling the 23 miles of aisles at Comdex Fall, looking for programmer tools, is like sifting the sand hills over in Lockhart Gulch west of Scotts Valley, looking for sharks' teeth. You know that they're down there, and if you dig long enough you'll find a few. However, the smart guys run down to New Age Annie's Kosmic Krystal Koop in Santa Cruz and buy one of the nice clean sharks' teeth Annie keeps in a "Save the Whales" bowl next to the two-for-a-dollar tiger eyes. Saves a heap o' diggin' — which is what you're doing by buying this magazine.

Into the Outback

What wild and wonderful programmer stuff there is is not on the main floor, by and large. (Exceptions might include the Microsoft booth, which was the size of a small county in Arkansas.) Finding the good stuff means traipsing around the outlying hotels such as the Tropicana and Bally's.

The #1 Neat Comdex Idea for programmers comes from two different vendors, who solved the same knotty problem using two different technologies. The problem is a common one: Running out of DOS memory while doing a build on a large application using command-line compilers and linkers. QuickPascal has this problem in spades; for all its many virtues, QP uses memory like cheap cologne and always runs out before Turbo Pascal. Even a memory miser such as Turbo will run out eventually if you hand it a big enough application.

Jeff Duntemann, KI6RA

Qualitas' superb 386-to-the-MAX nibbles on the problem by using the 386's hardware memory manager to remap some of 386 extended memory down beneath the video refresh buffer. You can get a contiguous DOS memory area as large as 704K if you're using a monochrome display adapter. A small San Jose, California company named V Communications takes the idea much further, by moving the video refresh buffer entirely to some other location in 386 memory and making BIOS aware of the move. Their Memory Commander product can give you as much as 924K of contiguous DOS memory, depending on what TSRs, device drivers, and BIOS software needs space in the first megabyte.

924K is an extreme case. The company says a typical system should be able to have about 860K available for compiles, if no attempt is made to address screen memory directly. Because command-line compilers and linkers typically write to standard output rather than the refresh buffer, this is not a problem. And 860K could allow you to build a much larger app. Think of all that symbol table space . . .

Invisible Software of Foster City, Calif. has a product that does much the same thing, only they use a little-known and less-understood feature called "shadow RAM," supported by several of the Chips and Technologies VLSI chip sets for 286 and 386 motherboards. Shadow RAM is present only in those machines using those chip sets. If the motherboard is equipped with a minimum 1 Mbyte of RAM, (rather than the canonical 640K) the chip set can map portions of that RAM where it needs to. The feature was developed to allow the copying of code from slower BIOS ROMs into faster RAM to improve performance, but it can also map RAM into the segment space between $A000 and $B800 (assuming you don't have a monochrome display board) giving you a contiguous DOS space of as much as 736K. So while the Invisible RAM product does not give you quite as much potential space as Memory Commander, it has the advantage of working in the great many inexpensive Asian 286 motherboards that use the Chips chip sets. (Memory Commander, remember, is a 386-only product.) You can download a test program from Invisible Software's BBS to detect and report on whether you have the necessary chip set in your system. Call them for details if you're interested; it's a very slick product.

Documentation on Demand

The #2 Neat Comdex Idea for programmers solves an ugly logistical problem facing shareware authors: How to provide attractive printed documentation without going broke. As one of the inducements to registering a shareware package, many authors offer typeset printed documentation. The catch is that manuals cannot be printed economically in batches of fewer than 500 or so, and costs don't really go down until the numbers head up into the tens of thousands.

However, when you punt your shareware creation out into the brave, cold world, you have no idea how many registrations you're likely to get. Worse, products generally evolve far more quickly than 500 manuals are likely to be needed, leaving authors stuck with piles of obsolete manuals that are fully paid for — and worthless.

Workhorse laser printers (especially HP's that prints on both sides of a sheet at once) and desktop publishing packages such as Ventura Publisher allow high-quality, short-run printed output. What's needed is a mechanism to bind loose sheets together in a professional-looking way, and at Comdex I found one: The Unibind binding system.

In a nutshell, Unibind works like this: The sheets to be bound are placed inside a plastic or card-stock folder with a thermoplastic adhesive bar running down the middle. This assemblage is then placed in a toaster-gadget that positions the sheets and cover accurately, and heats them until the adhesive melts and glues the sheets together at the spine and the spine to the cover. The system can bind stacks from 2 sheets to 650 sheets in size, and each volume takes about 45 seconds to bind.

Systems similar to this have been available for some time, but the ones I've seen and used (typically from Cheshire) are extremely messy and mechanically fragile. Unibind is neither; the bound volumes are tidy and show no loose traces of adhesive, and the binder device has far fewer moving parts than Cheshire and similar systems.

Once bound, the sheets are in there for the long haul; I was unable to pull any of the sheets from the bound volume without tearing them. On the downside, the system has significant upfront costs, and the per-piece cost of the bound volumes is higher than volumes printed and bound at a printing plant. However, there is no waste and no obsolescence, because the system truly allows "documentation on demand." You print what you need as you need it, folding in updates as they happen, no sooner, no later. You can support several low-volume shareware products without going broke printing 500 manuals for each while expecting to sell maybe 20 or 30 manuals per year.

It's getting tougher and tougher all the time to put low-cost specialty software products on the market and make them pay. Shareware is our last best hope in this regard, and Unibind can help solve that ugly documentation issue. If you're a shareware author you ought to look into it.

Stereo-On-A-Card

The #3 Neat Comdex Idea for programmers may seem a little loopy, but it solved an infuriating problem for me and may solve that same problem for you if you're one of the many programmers who listens to music while programming. The Desktop Stereo product from Optronics of Ashland, Ore. is a half-sized board for the PC bus containing a world-class FM stereo receiver and 4 watts per channel amplifier. There are no tuning knobs on the board bracket; all controls are done electronically, through pop-up dialog boxes containing, among other things (dare I say it?) radio buttons. You can view the FM band as a graph of vertical bars displaying signal intensity at various frequencies (neat touch!) and preset up to ten frequencies with mnemonic names such as "KRAP" or "Hillbilly Rock" and punch them up like buttons on your car radio.

The problem that this board solves is that the expensive Japanese CD-equipped boom boxes that many of us place beside our RAM charged 386 boxes leak like sieves. Unless your favorite FM station's towers are on the next block, what you'll hear on your FM receiver is likely to be your machine's switching transients playing solo, and that is dull (if powerful) music. I'd long since abandoned FM and simply play my CDs. The FM module on the Desktop Stereo card is extremely well shielded (it had better be!) and absolutely quiet in the absence of signal modulation.

Now I can listen to PBS again. 20 plus stations accessible from fringey Scotts Valley. No racket. Jeff-Bob says check it out.

All Set with Modula

Let's continue our discussion of the vice president of Structured Languages, Modula-2. Will Modula ever overtake Pascal for small machines? Probably not. Unless . . . the president decides not to run in OS/2 land, in which case the race gets interesting. Modula-2 is already very big over on the OS/2 side of things, second (so far as I can tell) only to You Know What. If this continues for a few more years, the OS/2 products could achieve a formidable critical mass, especially since Modula contains standard syntactic support for multitasking. (More on that very thorny issue when I get OS/2 running reliably on this sorry excuse for a 386 machine.) If you're contemplating a project for OS/2, ignore those C-sirens claiming that C is the only way to go. You can do very well with Modula-2, according to sources that I trust. Someday I'll know from firsthand experience, sigh.

No, in this issue we're going to talk about sets. Sets are what drove me out of Modula-2 several years ago. When the language spec was first released I jumped on it, with full intent to port over my disks full of code, written in

the faltering corpse of Pascal/MT+ for CP/M-80. I dug in and discovered several days into the project that I couldn't do it. My code was absolutely peppered with the killer type definition:

```
TYPE
   CharSet = SET OF Char;
```

Uh-uh, said the compiler. Sets in Modula-2 may have no more than 16 elements.

This is a serious semantic bite in the buns. Sets work well for me and I use them a lot, especially for building systems to handle characters moving from one place to another, as from the keyboard to the screen or from a serial port to the screen or to a disk file. Like Maxwell's Demon, a set is a filter that can pass odd characters among the ASCII throng while denying passage to others in a group just as odd. Consider the elegance of this classic construct:

```
IF AnswerChar IN ['Y','y']
   THEN DoIt ELSE DontDoIt;
```

The alternative is this:

```
IF (AnswerChar = 'Y') OR
   (Answer-Char = 'y')
   THEN . . .
```

You might argue that the second form resolves to fewer machine instructions, and I'd argue back that you're rarely going to have to execute 17,000 such tests in a tight loop. Furthermore, what about this:

```
IF IncomingChar IN WhiteSpaceSet
                          THEN . . .
```

There's simply nothing like sets for character filters such as this. It was just possibly possible in some cases to pull tricks with subranges of fewer than 16 characters, but the whole notion offended me: Niklaus Wirth threw character sets out the window to make it easier to implement Modula-2. There are maybe two or three hundred potential Modula-2 compiler implementors in this world. There are hundreds of thousands of potential Modula-2 programmers. One suspects he skipped Marketing 101 as an undergrad.

About then Turbo Pascal happened, and Modula-2 slipped into eclipse for some years. Logitech held the torch alight all that time, but their product, while solid, was complex and slow and admittedly intended for internal use. It wasn't until JPI introduced TopSpeed Modula-2 that the language showed any serious life. Soon afterward, the Stony Brook compiler made its debut, and I've begun to do some serious work in Modula again.

The reason is pretty simple: TopSpeed and Stony Brook have done the Awful Thing: Extended Modula-2 by allowing sets to have as many as 65,536 elements. Horrors. You might not be able to port your dog kennel management package to the Lilith operating system. It is to cry real tears.

Niklaus Wirth threw character sets out the window

Duntemann's One Law of Portability

Remember this, chilluns: For any platform with I/O more complex than a batch system, semantic differences between platforms makes portability impossible. In other words, even if you wrote your character-based PC kennel manager in absolutely standard Modula-2, could you port it to the Macintosh? If you had written it for multiple terminals under Unix, could you port it to DOS? Get real — the effort spent resolving semantic conflicts would far outweigh trifles like the shape of an IF statement.

So let's quit arguing about something that's never been worth a plugged nickel outside of academe anyway.

Watch the Corral, Not the Cows!

A set is an abstraction of a group of values, indicating whether one or more of those values are present or not present. It's like a corral on a farm with seven cows; at any given time a cow is either in the corral or not. The cows are in no particular order within the corral. They're either there or else out making things for the unwary to step in.

It's important to remember that the set is not the cows; the set is the corral. It's still a set even when it is empty.

In Modula-2, a set is defined in terms of some ordinal type or subrange of an ordinal type, including enumerations such as the insufferable list of colors that every writer on the subject (myself included) has used in books explaining the concept:

```
TYPE
   Colors = (Red, Orange,
   Yellow, Green, Blue, Indigo,
                          Violet);
   WarmColors = [Red . . Yellow];
   ColorSet = SET OF Colors;
   WarmSet = SET OF WarmColors;
   CardSet = {0..65535}
   CharSet = SET OF CHAR; (* Yay! *)
```

Beneath it all, in physical memory, a set is a bitmap. There is one bit in the set for each value that may legally be present in the set. Each bit carries one Boolean fact: Whether the value that the bit stands for is present or not present in the set. Adding a value to the set is done by raising that value's bit to binary 1. Removing a value from the set is done by changing that value's bit back to a binary 0.

A "full" set (that is, one having all values present) is not one bit larger than an empty set. Again, the set is the corral, not the cows!

Set Operators

There are a number of operators and standard procedures that work on sets in Modula-2. The two most obvious are *INCL*, which places a value in a set, and *EXCL*, which removes a value from a set. These are not present in Pascal. *IN* is still there, doing exactly what it does in Pascal: Return a Boolean value indicating whether the value on the left is present in the set on the right. Ditto >= (set inclusion, right in left), and <= (set exclusion, left in right) which do much the same but for whole sets: >= returns TRUE if all values of the set on its right are present in the set on its left; and <= returns TRUE if all values in the set on its left are present in the set on its right.

There are actually only four operators that are true set operators in that they act on sets and return sets: + (set union) – (set difference) * (set intersection) and / (set symmetric difference). Of these, only the first three are present in Pascal.

Set union of two sets returns the set that contains all the elements present in both of the sets taken as one. Set intersection of two sets returns the set of values that are present in both sets, but none of those values that may be present in one or the other but not both.

Set difference is a little trickier; my Pascal prof explained it badly (getting it mixed up with symmetric difference, see below) and I misunderstood it through ten years and two editions of my book. Set difference of two sets returns the set that consists of the elements in the set on the left once those in the set on the right have been removed from it.

Basically, set difference is a way of pulling several elements out of a set without using *EXCL* to do it one element at a time:

(continued from page 136)

{'A'..'Z'} – {'M'..'Z'}

This set expression returns the set *{'A'..'L'}*. (Keep in mind that Modula-2 uses curly brackets for set constructors rather than straight brackets.)

Finally, set symmetric difference (which is not in any Pascal implementation I'm aware of) is rather like set union turned inside out. The symmetric difference of two sets is the set of all elements that are present in one or the other set, but not in both sets. In a sense, the symmetric difference of two sets is what the two sets don't have in common; for example, what remains once their intersection (overlap) has been removed.

Remember that the set is not the cows; the set is the corral

Among them, these operators allow you to do just about anything with a set that you'd ever want to do. And now that sets can have up to 65,535 elements in Modula-2, that's a lot.

The Naked Set

Wirth's original language definition did not hard-code 16 as the number of elements in a set. The number of elements in a Modula-2 set was originally defined as the number of elements in the machine word used by the system for which the compiler was implemented. In other words, in a system with a 32-bit word there would be 32 possible elements in a Modula-2 set.

This makes those limited set operations very easy to implement, and very fast, because they can be done using the native bit-manipulation instructions present in all modern-day CPUs. Remember that sets are bitmaps. Furthermore, the four true set operators bear a certain uncanny functional resemblance to certain logical operators such as *AND*, *OR*, and *XOR*.

OR the bits of two sets together and whammo, suddenly you have the union of the two sets. *AND* the bits of two sets together, and what remains is the intersection of the two sets. *AND* the bits of one set with the complement (reversed) bits of another set, and you remove the bits of the complemented set from the other set, that is, set difference. Finally, *XOR* the bits in two sets together and what's left are the bits that are present in one set or the other but not in both sets, since *XOR* drives identical bit pairs to 0. Voilà: Symmetric set difference.

This is, of course, exactly what Wirth intended, and he intended for it all to happen within the accumulator of the host CPU, ensuring speed and minimal fussing. Happily, in this brave new world of fast global optimizing compilers (Stony Brook's is fabulous) we can have it both ways: When we're fiddling small sets we can do it fast at one shot inside the accumulator; when we're fiddling big sets we can do it a word at a time and take the performance hit.

Now, Wirth defined a specific kind of set that has no true analog in Pascal: *BITSET*, a standard type supported in all Modula-2 compilers. A *BITSET* is a machine word used as a bitmap. All of the set operators operate on *BITSET* values. A *BITSET*'s nominal values are 0 . . 15, but these are bit numbers more than values. A *BITSET* is thus a sort of naked set, in which the bitmap nature of the set is laid bare and can be manipulated directly. A bit in a *BITSET* does not abstract a color, or a character, or a cardinal number, or a cow; a bit in a *BITSET* represents a bit, period.

Twiddling Bits in Other Types

With very little futzing, this fills an apparent gap in Modula-2: The lack of explicit bit-manipulation facilities. Turbo Pascal has explicit bitwise *AND*, *OR*, *NOT*, and *XOR* operators for numeric ordinal types, and it can also shift bits in numeric ordinal values with its *SHR* and *SHL* operators. Modula-2 has none of these . . . or does it?

It does . . . but they only operate on values of type *BITSET*.

No problem — just ask Pizza Terra. (For those unfamiliar with the reference, see my May 1989 column.) Modula-2 has explicit type casting (which Wirth calls type coercion), so if you want to fiddle bits in type *CHAR*, cast type *CHAR* onto type *BITSET*, and fiddle away! Any type can be cast onto any other type of identical size, and there are transfer functions such as *Ord* to cast 8-bit types like *CHAR* and *BOOLEAN* onto 16-bit types like *CARDINAL*.

For example, to *AND* a *CARDINAL* variable *MyCard* with the value 128, you could do this:

```
NewCard :=
   CARDINAL(BITSET
         (MyCard) * BITSET(128));
```

Here, *MyCard* and the value 128 are

(continued on page 141)

(continued from page 138)
both cast onto *BITSET*s, which are then *AND*ed together by using the set intersection operator, which is equivalent (on a bit level) to *AND*. Finally, the result of the set intersection operation is cast back onto a *CARDINAL* for assignment to the *CARDINAL* variable *NewCard*.

This works . . . but it sure as hell isn't obvious. Unfortunately, in Modula this is how the game is played. Better to disguise all this arm-twisting of types (coercion is such a lovely word!) behind some procedures with more mnemonic names. This is what I've done in the listings for this column, which present a Modula-2 module called *Bitwise*. Listing One, page 150, is the definition module for *Bitwise*, and Listing Two, page 150, is the implementation module.

Bitwise provides function procedures to perform bitwise *AND*, *OR*, *XOR*, and *NOT* operations. (See Table 1.) Note that the capitalization is different from that used here in the descriptive text, in order to differentiate my procedure *And* from the existing (and incompatible) Boolean logical operator *AND*. (Case is significant in Modula-2, and this is the first time in my career I've caught myself being glad. Crazy world, ain't it?) Additionally, *Bitwise* contains procedures to set, clear, and test individual bits, and also to shift values right or left by up to 16 bits. This suite of routines provides roughly the same bit-banging power you get stock in Turbo Pascal. This seems to be the lot of Modula-2 programmers: To perpetually build what those Turbo guys have come to take for granted!

The formal parameters for all of the routines in *Bitwise* are type *CARDINAL*, because *CARDINAL* is the unsigned 16-bit numeric type in Modula-2, equivalent to *Word* in Turbo Pascal. It's a good basic foundation upon which to cast all other ordinal types in Modula-2. (And it's used quite a bit by itself.) If you want to set bit number 3 in a character, for example, you could do this:

```
NewChar :=
        CHAR(SetBit(ORD('A'),3));
```

The *ORD* transfer function casts the character value onto a *CARDINAL* value for passing to the *SetBit* function procedure, and finally the *CARDINAL* value returned by *SetBit* is cast back onto a character for assignment to *NewChar*.

Read over the code implementing *Bitwise* and it all makes sense to you. Again, understand type casting/coercion and you've got it in your hip pocket.

When Words Runneth Over

There is something a little bit hazardous about *Bitwise*. The *SHR* and *SHL* routines can cause overflow errors if you shift bits to the extent that 1-bits roll out of either side of the 16-bit word in which they exist. Stony Brook Modula-2 code checks for overflow errors and will crash your program when you shift bits out of the word they live in.

Now, shifting bits off the edge of their word is not necessarily a bad thing. Sometimes you do it deliberately to get rid of the bits in question. There's nothing inherently damaging about it, because on a machine level the bits get shunted first into the carry flag and then off into nothingness. (What we affectionately call "the bit bucket.") Adjacent data is never overwritten, no matter if we try to shift a bit by (a meaningless) 245 positions.

The way out is to turn off overflow error checking. Enter here one of my major arguments with Modula-2: For portability's sake (gakkh!) there are no compiler toggles. Turbo Pascal has a whole raft of them, things like *{$R-}* and so on. The situation would seem to call for bracketing the *SHR* and *SHL* routines between compiler toggles that switch overflow checking off only for the duration of the routine, then on again once the routine terminates.

Sorry, Charlie. As every good tuna fish knows, compiler toggles are implementation dependent and destroy the prospects for portability. Lord knows, we can't have that, now, can we? The best that can be done with the Stony Brook compiler is to turn off overflow checking entirely within the *Bitwise* module by changing the compile options on a by-module basis. Be sure to do this when you compile and use *Bitwise*! If you're using a Modula compiler in which overflow checking cannot be turned off, you'd better add

Bitwise operators	Set operation	
AND	*	Intersection
OR	+	Union
XOR	/	Symmetric difference
NOT	{0..15} - BITSET	"Full" set - target set

Table 1: *Relating bitwise operators to set operations*

safety belts to any code that uses *SHL* and *SHR*.

The Boss DOS Book

There is a certain type of book I call a "category killer;" it's *the* book on a certain subject and tends to keep other books of its type from being published. One of these is Ray Duncan's *Advanced MS-DOS* (Microsoft Press), a book that has never been very far from my left hand while sitting in this particular chair. I'm pleased to report that Ray has company, in the form of Que Corporation's *DOS Programmer's Reference*, by Terry Dettmann. On 892 pages Terry has managed to summarize every BIOS function through PS/2, every DOS call through V4.0, all mouse function calls, all EMS function calls, and a blizzard of other information including low-level disk structure, device driver and interrupt programming, serial port programming, and lots more.

The very best part about this book, however, may well be its index. Having 892 pages of information is small comfort if you can't find anything when you need it in a hurry. The index occupies 33 pages, with about 100 citations per page, set small in two columns. Everything I tried to look up was either indexed or not covered in the book. (And things that weren't covered really shouldn't have been anyway, like VGA hardware architecture details.)

Altogether, the best hacker's book to cross my desk in a good long while. Get it.

Products Mentioned

Memory Commander
V Communications
3031 Tisch Way, Ste. 802
San Jose, CA 95128
408-296-4224
$129.95

Invisible RAM
Invisible Software
1165 Chess Drive, Ste. D
Foster City, CA 94404
415-570-5967
$39.95

Unibind
Unibind Systems
7900 Capwell Drive
Oakland, CA 94621
415-638-1060
Various configurations and prices
Contact the vendor for specifics

Desktop Stereo
Optronics Technology
P.O. Box 3239
Ashland, OR 97520
503-488-5040
$199

DOS Programmer's Reference, 2nd edition
Terry Dettmann, revised by Jim Kyle
Que Corporation, 1989
ISBN 0-88022-458-4
Softcover, 892 pages, $27.95

Dredging the Channel

There are millions — nay, tens of millions — of DOS machines out there, and various research reports I've seen indicate that the greatest growth potential lies in machines of modest cost and capabilities: The "bare bone" 88 and 286 clones that fill *Computer Shopper* to a depth of 800+ pages every month. There are already 30 million of them (conservative estimate) and in another few years there could be as many as 100 million of them out there, plugging away. This is an utterly unbelievable market for software products, and yet the distribution channel has closed up to the point that a small-time operator (like most of us) has no chance to make those millions of people even aware of the existence of their products.

There has got to be a way. Any ideas? Pass them by me. I'll be talking about this subject in future months, and I'll share some guerrilla marketing concepts I've devised, and will discuss how the little guys can shove some very big rear ends out of their monopoly position in the retail channel.

Write to Jeff Duntemann on MCI Mail as JDuntemann, or on CompuServe to ID 76117,1426.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 150.)

Vote for your favorite feature/article.
Circle Reader Service **No. 12.**

Listing One *(Text begins on page 127.)*

```
/* ------------ dir.h ----------- */

/* Substitute Lattice directory functions for
 * Turbo C directory functions
 */

#include <dos.h>

#define ffblk FILEINFO
#define ff_name name

#define findfirst(path,ff,attr) dfind(ff,path,attr)
#define findnext(ff) dnext(ff)
```

End Listing One

Listing Two

```
/* ---------- search.c ----------- */

/*
 * the TEXTSRCH retrieval process
 */

#include <stdio.h>
#include <string.h>
#include <curses.h>
#include "textsrch.h"

static char fnames[MAXFILES] [65];
static int fctr;

static void select_text(void);
static void display_page(WINDOW *file_selector, int pg);
void display_text(char *fname);

/* ---- process the result of a query expression search ---- */
void process_result(struct bitmap map1)
{
    int i;
    extern int file_count;
    for (i = 0; i < file_count; i++)
        if (getbit(&map1, i))
            strncpy(fnames[fctr++], text_filename(i), 64);
    initscr();                  /* initialize curses */
    select_text();              /* select a file to view */
    endwin();                   /* turn off curses */
    fctr = 0;
}

/* ------- search the data base for a word match -------- */
struct bitmap search(char *word)
{
    struct bitmap map1;

    memset(&map1, 0xff, sizeof (struct bitmap));
    if (srchtree(word) != 0)
        map1 = search_index(word);
    return map1;
}

#define HEIGHT 8
#define WIDTH 70
#define HOMEY 3
#define HOMEX 3

#define ESC 27

/* --- select text file from those satisfying the query ---- */
static void select_text(void)
{
    WINDOW *file_selector;
    int selector = 0; /*selector cursor relative to the table */
    int cursor = 0;   /*selector cursor relative to the screen*/
    int keystroke = 0;

    /* --- use a window with a border to display the files -- */
    file_selector = newwin(HEIGHT+2, WIDTH+2, HOMEY, HOMEX);

    keypad(file_selector, 1);       /* turn on keypad mode    */
    noecho();                       /* turn off echo mode     */
    wsetscrreg(file_selector, 1, HEIGHT);/* set scroll limits */

    /* -------- display the first page of the table --------- */
    display_page(file_selector, 0);

    while (keystroke != ESC)    {
        /* ----- draw the window frame ------ */
        box(file_selector, VERT_DOUBLE, HORIZ_DOUBLE);

        /* ------------ fill the selector window ------------ */
        mvwaddstr(file_selector, cursor+1, 1, "->");
        wrefresh(file_selector);

        /* -------------- make a selection ------------------- */
        keystroke = wgetch(file_selector);/* read a keystroke */
        mvwaddstr(file_selector, cursor+1, 1, "  ");
        switch (keystroke)  {
            case KEY_HOME:
                /* -------- Home key (to top of list) ------- */
                selector = cursor = 0;
                display_page(file_selector, 0);
                break;

            case KEY_END:
                /* ------ End key (to bottom of list) ------ */
                selector = fctr - HEIGHT;
                if (selector < 0)   {
                    selector = 0;
                    cursor = fctr-1;
                }
                else
                    cursor = HEIGHT-1;
                display_page(file_selector, selector);
                break;

            case KEY_DOWN:
                /* - down arrow (move the selector cursor) -- */
                /* --------- test at bottom of list --------- */
                if (selector < fctr-1)  {
                    selector++;
                    /* ------ test at bottom of window ------ */
                    if (cursor < HEIGHT-1)
                        cursor++;
                    else    {
                        /* ---- scroll the window up one ---- */
                        scroll(file_selector);
                        /* --- paint the new bottom line ---- */
                        mvwprintw(file_selector, cursor+1, 3,
                            fnames[selector]);
                    }
                }
                break;

            case KEY_UP:
                /* --- up arrow (move the selector cursor) -- */
                /* ----------- test at top of list ---------- */
                if (selector)   {
                    --selector;
                    /* -------- test at top of window ------- */
                    if (cursor)
                        --cursor;
                    else    {
                        /* --- scroll the window down one --- */
                        winsertln(file_selector);
                        /* ----- paint the new top line ----- */
                        mvwprintw(file_selector, 1, 3,
                            fnames[selector]);
                    }
                }
                break;

            case '\n':
                /* -- user selected a file, go display it --- */
                display_text(fnames[selector]);
                break;

            case ESC:
                /* --------- exit from the display ---------- */
                break;

            default:
                /* ----------- invalid keystroke ------------ */
                beep();
                break;
        }
    }
    delwin(file_selector);  /* delete the selector window */
    clear();                /* clear the standard window  */
    refresh();
}

/* ------ display a page of the file selector window ------- */
static void display_page(WINDOW *file_selector, int line)
{
    int y = 0;
    werase(file_selector);
    while (line < fctr && y < HEIGHT)
        mvwprintw(file_selector, ++y, 3, fnames[line++]);
}
```

End Listing Two

Listing Three

```
/* ---------------- display.c ----------------- */

/* Display a text file on the screen.
 * User may scroll and page the file.
 * Highlight key words from the search.
 * User may jump to the next and previous key word.
 */

#include <stdio.h>
#include <stdlib.h>
#include <curses.h>
#include <ctype.h>
#include <string.h>
#include "textsrch.h"
#define ESC 27

/* ----------- header block for a line of text ----------- */
struct textline {
```

(continued on page 146)

Listing Three *(Listing continued, text begins on page 127.)*

```
    char keys[5];               /* offsets to key words     */
    struct textline *nextline;  /* pointer to next line     */
    char text;                  /* first character of text  */
};

/* --------- listhead for text line linked list -------- */
struct textline *firstline;
struct textline *lastline;

int pagemarked(int topline);
static void do_display(FILE *fp);
static void findkeys(struct textline *thisline);
static void display_textpage(WINDOW *text_window, int line);

/* ---------- display the text in a selected file --------- */
void display_text(char *filepath)
{
    FILE *fp;

    fp = fopen(filepath, "r");
    if (fp != NULL) {
        do_display(fp);
        fclose(fp);
    }
    else    {
        /* ----- the selected file does not exist ----- */
        char ermsg[80];
        sprintf(ermsg, "%s: No such file", filepath);
        error_handler(ermsg);
    }
}

static void do_display(FILE *fp)
{
    char line[120];
    WINDOW *text_window;
    int keystroke = 0;
    int topline = 0;
    int linect = 0;
    struct textline *thisline;

    firstline = lastline = NULL;

    /* --------- read the text file into the heap ------- */
    while (fgets(line, sizeof line, fp) != NULL)    {
        line[78] = '\0';
        thisline =
            malloc(sizeof(struct textline)+strlen(line)+1);
        if (thisline == NULL)
            break;      /* no more room */

        /* ----- clear the text line record space -------- */
        memset(thisline, '\0', sizeof(struct textline) +
                        strlen(line)+1);

        /* ---- build the text line linked list entry ---- */
        if (lastline != NULL)
            lastline->nextline = thisline;
        lastline = thisline;
        if (firstline == NULL)
            firstline = thisline;
        thisline->nextline = NULL;
        strcpy(&thisline->text, line);

        /* ------------ mark the key words ------------ */
        findkeys(thisline);
        linect++;
    }

    /* ------- build a window to display the text ------- */
    text_window = newwin(LINES, COLS, 0, 0);
    keypad(text_window, 1);     /* turn on keypad mode    */

    while (keystroke != ESC)    {
        /* --- display the text and draw the window frame --- */
        display_textpage(text_window, topline);
        box(text_window, VERT_SINGLE, HORIZ_SINGLE);
        wrefresh(text_window);

        /* ------------ read a keystroke ------------- */
        keystroke = wgetch(text_window);
        switch (keystroke)  {
            case KEY_HOME:
                /* ------- Home key (to top of file) ------ */
                topline = 0;
                break;
            case KEY_DOWN:
                /* --- down arrow (scroll up) ---- */
                if (topline < linect-(LINES-2))
                    topline++;
                break;
            case KEY_UP:
                /* ----- up arrow (scroll down) ---- */
                if (topline)
                    --topline;
                break;
            case KEY_PGUP:
                /* -------- PgUp key (previous page) -------- */
                topline -= LINES-2;
                if (topline < 0)
                    topline = 0;
                break;
            case KEY_PGDN:
                /* -------- PgDn key (next page) ------------ */
                topline += LINES-2;
                if (topline <= linect-(LINES-2))
                    break;
            case KEY_END:
```

(continued on page 148)

Listing Three *(Listing continued, text begins on page 127.)*

```
                /* ------- End key (to bottom of file) ------ */
                topline = linect-(LINES-2);
                if (topline < 0)
                    topline = 0;
                break;
            case KEY_RIGHT:
                /* - Right arrow. Go to next marked key word  */
                do  {
                    /* -- repeat PGDN until we find a mark -- */
                    topline += LINES-2;
                    if (topline > linect-(LINES-2)) {
                        topline = linect-(LINES-2);
                        if (topline < 0)
                            topline = 0;
                    }
                    if (pagemarked(topline))
                        break;
                }   while (topline &&
                        topline < linect-(LINES-2));
                break;
            case KEY_LEFT:
                /* Left arrow. Go to previous marked key word */
                do  {
                    /* -- repeat PGUP until we find a mark -- */
                    topline -= LINES-2;
                    if (topline < 0)
                        topline = 0;
                    if (pagemarked(topline))
                        break;
                }   while (topline > 0);
                break;
            case ESC:
                break;
            default:
                beep();
                break;
        }
    }
    /* -------- clean up and exit --------- */
    wclear(text_window);
    wrefresh(text_window);
    delwin(text_window);
    thisline = firstline;
    while (thisline != NULL)    {
        free(thisline);
        thisline = thisline-> nextline;
    }
}

/* ---- test a page to see if a marked keyword is on it ---- */
int pagemarked(int topline)
{
    struct textline *tl = firstline;
    int line;
    while (topline-- && tl != NULL)
        tl = tl->nextline;
    for (line = 0; tl != NULL && line < LINES-2; line++)    {
        if (*tl->keys)
            break;
        tl = tl->nextline;
    }
    return *tl->keys;
}

#define iswhite(c) ((c)==' '||(c)=='\t'||(c)=='\n')

/* ---- Find the key words in a line of text. Mark their
   character positions in the text structure ------- */
static void findkeys(struct textline *thisline)
{
    char *cp = &thisline->text;
    int ofptr = 0;

    while (*cp && ofptr < 5)    {
        struct postfix *pf = pftokens;/* the query expression */
        while (iswhite(*cp))  /* skip the white space */
            cp++;
        if (*cp)    {
            /* ---- test this word against each argument in the
                query expression ------- */
            while (pf->pfix != TERM)    {
                if (pf->pfix == OPERAND &&
                        strnicmp(cp, pf->pfixop,
                            strlen(pf->pfixop)) == 0)
                    break;
                pf++;
            }
            if (pf->pfix != TERM)
                /* ----- the word matches a query argument.
                   Put its offset into the line's header --- */
                thisline->keys[ofptr++] =
                    (cp - &thisline->text) & 255;
            /* --- skip to the next word in the line --- */
            while (*cp && !iswhite(*cp))
                cp++;
        }
    }
}

/* --- display page of text starting with specified line --- */
static void display_textpage(WINDOW *text_window, int line)
{
    struct textline *thisline = firstline;
    int y = 1;

    wclear(text_window);
    wmove(text_window, 0, 0);
```

```
    /* ---- point to the first line of the page ----- */
    while (line-- && thisline != NULL)
        thisline = thisline->nextline;

    /* ------- display all the lines on the page ------ */
    while (thisline != NULL && y < LINES-1) {
        char *cp = &thisline->text;
        char *kp = thisline->keys;
        char off = 0;
        wmove(text_window, y++, 1);

        /* ------ a character at a time -------- */
        while (*cp) {
            /* --- is this character position a key word? --- */
            if (*kp && off == *kp)  {
                wstandout(text_window); /* highlight key words*/
                kp++;
            }

            /* ---- is this character white space? ---- */
            if (iswhite(*cp))
                wstandend(text_window); /* turn off hightlight*/

            /* ---- write the character to the window ------ */
            waddch(text_window, *cp);
            off++;
            cp++;
        }
        /* -------- a line at a time ---------- */
        thisline = thisline->nextline;
    }
}
```

End Listing Three

Listing Four

```
/* ------------ error.c ------------ */

/* General-purpose error handler */

#include <curses.h>
#include <string.h>

void error_handler(char *ermsg)
{
    int x, y;
    WINDOW *error_window;

    x = (COLS - (strlen(ermsg)+2)) / 2;
    y = LINES/2-1;
    error_window = newwin(3, 2+strlen(ermsg), y, x);
    box(error_window, VERT_SINGLE, HORIZ_SINGLE);
    mvwprintw(error_window, 1, 1, ermsg);
    wrefresh(error_window);
    beep();
    getch();
    wclear(error_window);
    wrefresh(error_window);
    delwin(error_window);
}
```

End Listings

Listing One *(Text begins on page 134.)*

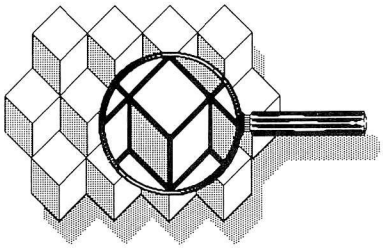
```
(*-------------------------------------------------------*)
(*                  BITWISE.MOD                          *)
(*              Definition Module                        *)
(*                                                       *)
(*     Bit-manipulation routines for Modula-2            *)
(*                                                       *)
(*                            by Jeff Duntemann          *)
(*                            For DDJ : March 1990       *)
(*                            Last modified 11/25/89     *)
(*-------------------------------------------------------*)

DEFINITION MODULE Bitwise;

PROCEDURE And(A,B : CARDINAL) : CARDINAL;

PROCEDURE Or(A,B : CARDINAL) : CARDINAL;

PROCEDURE Xor(A,B : CARDINAL) : CARDINAL;

PROCEDURE Not(Target : CARDINAL) : CARDINAL;

PROCEDURE SetBit(Target : CARDINAL; BitNum : CARDINAL) : CARDINAL;

PROCEDURE ClearBit(Target : CARDINAL; BitNum : CARDINAL) : CARDINAL;

PROCEDURE TestBit(Target : CARDINAL; BitNum : CARDINAL) : BOOLEAN;

PROCEDURE SHR(Target : CARDINAL; By : CARDINAL) : CARDINAL;

PROCEDURE SHL(Target : CARDINAL; By : CARDINAL) : CARDINAL;

END Bitwise.
```

End Listing One

Listing Two

```
(*-------------------------------------------------------*)
(*                  BITWISE.MOD                          *)
(*            Implementation Module                      *)
(*                                                       *)
(*     Bit-manipulation routines for Modula-2            *)
(*                                                       *)
(*                            by Jeff Duntemann          *)
(*                            For DDJ : March 1990       *)
(*                            Last modified 11/25/89     *)
(*                                                       *)
(*  NOTES ON THE CODE:                                   *)
(*                                                       *)
(*  In all cases below, BitNum MOD 16 is used as a       *)
(* means of ensuring that BitNum will be in the          *)
(* range of 0..15.  MOD 16 divides by 16 but returns     *)
(* the remainder, which cannot be over 15 when you       *)
(* divide by 16.                                         *)
(*-------------------------------------------------------*)

IMPLEMENTATION MODULE Bitwise;

VAR
  I       : CARDINAL;
  TempSet : BITSET;


PROCEDURE And(A,B : CARDINAL) : CARDINAL;

BEGIN
  RETURN CARDINAL(BITSET(A) * BITSET(B));
END And;


PROCEDURE Or(A,B : CARDINAL) : CARDINAL;

BEGIN
  RETURN CARDINAL(BITSET(A) + BITSET(B));
END Or;


PROCEDURE Xor(A,B : CARDINAL) : CARDINAL;

BEGIN
  RETURN CARDINAL(BITSET(A) / BITSET(B));
END Xor;


PROCEDURE Not(Target : CARDINAL) : CARDINAL;

BEGIN
  RETURN CARDINAL({0..15} - BITSET(Target));
END Not;


PROCEDURE SetBit(Target : CARDINAL; BitNum : CARDINAL) : CARDINAL;

BEGIN
  TempSet := BITSET(Target);  (* INCL does not operate on expressions! *)
  INCL(TempSet,BitNum MOD 16);
  RETURN CARDINAL(TempSet);   (* Cast the target back to type CARDINAL *)
END SetBit;


PROCEDURE ClearBit(Target : CARDINAL; BitNum : CARDINAL) : CARDINAL;

BEGIN
  TempSet := BITSET(Target);  (* EXCL does not operate on expressions! *)
  EXCL(TempSet,BitNum MOD 16);
  RETURN CARDINAL(TempSet);   (* Cast the target back to type CARDINAL *)
END ClearBit;


PROCEDURE TestBit(Target : CARDINAL; BitNum : CARDINAL) : BOOLEAN;

BEGIN
  IF (BitNum MOD 16) IN BITSET(Target) THEN
    RETURN TRUE;
  ELSE
    RETURN FALSE;
  END;
END TestBit;


PROCEDURE SHR(Target : CARDINAL; By : CARDINAL) : CARDINAL;

BEGIN
  FOR I := 1 TO By DO
    Target := Target DIV 2;
  END;
  RETURN Target;
END SHR;


PROCEDURE SHL(Target : CARDINAL; By : CARDINAL) : CARDINAL;

BEGIN
  FOR I := 1 TO By DO
    Target := Target * 2;
  END;
  RETURN Target;
END SHL;


END Bitwise.
```

End Listings



Codecheck, a rule-based expert system that checks C and C++ source code for maintainability, portability, and compliance with in-house style, has been announced by **Conley Computing**. Codecheck has the ability to identify the number of operators per expression and lines per statement, and it provides a statistical analysis of code complexity and style, allowing programmers to check for both industry standards and those established by their company.

Codecheck also reviews code for its portability to ANSI C and K&R C, among others. Company president Patrick Conley told *DDJ* that Codecheck can be beneficial to both corporations and individuals, but especially to corporations that use many programmers for single projects. "The problem is getting programmers to adhere to standards; since everyone has their own Tower of Babel concerning standards, Codecheck can be programmed to check in-house style."

Codecheck supports all C compilers from major vendors, and is available for PC-DOS and Macintosh at $495, for OS/2 at $695, and for AIX, PC/IX, and QNX at $995. Multiple copy and educational discounts are also available. Reader service no 21.
Conley Computing
7033 SW Macadam Ave.
Portland, OR 97219
503-244-5253

The Paradox Engine, a C library for the relational database Paradox, has been announced by **Borland International**. The company claims that this product will enable C programmers to build applications that create or access Paradox data because programs that use the Paradox Engine are standard .EXE files. The benefit is interoperability among Borland's major business applications and languages, which theoretically allows the building of customized computing environments.

A program written with the Paradox Engine is compiled in C and linked with the Paradox Engine library to build an executable application that can dynamically access Paradox data. The PAL language can also access Paradox tables.

The engine provides an API of more than 70 funcitons, which allows the manipulation of Paradox tables in single and multiuser environments. The C version should be shipping this quarter, and will cost $495. A Pascal version is scheduled for release sometime in the middle of the year, and OS/2 and Windows versions are also under development. During the first 90 days of availability, registered Borland users can purchase the product for $195. Reader service no. 22.
Borland International
P.O. Box 660001
Scotts Valley, CA 95066-0001
408-439-1622

VRTX-PC, a real-time environment for the PC/XT/AT compatibles that allows these machines to be used as both development platforms and embedded computers, has been introduced by **Ready Systems**. Time-critical applications in which deterministic operating system performance is necessary can now be controlled by PCs. The company is excited that the VRTX-PC allows simultaneous development and execution of real-time multitasking applications, eliminating the need for low-level hardware control on the PC. They believe that this technology will reduce development costs and get products on the shelf faster.

The VRTX-PC real-time operating system supports MS-DOS functions, including all MS-DOS file and device I/O, and can be executed as a DOS resident program.

VRTX-PC includes a real-time kernel, a real-time debugger, an input/output file executive, a run-time library, a PC support executive, and a window manager that provides a user interface. For application development, VRTX-PC supports Microsoft C and Borland Turbo C. The price for a single user is $7600. Reader service no. 23.
Ready Systems
P.O. Box 60217
Sunnyvale, CA 94086
408-736-2600

The Sierra C toolset for the M68000 is available from **Sierra Systems**. The toolset includes an optimizing C compiler and complete C run-time library, two assemblers, linker, librarian, code management and debugging utilities, a serial downloader, a high-speed parallel downloader, and a source-level debugger. The company claims that the code produced is position independent, ROMable, and re-entrant.

The Sierra C compiler that is included in the toolset is ANSI compatible and supports the keywords and functionality required for embedded systems programming.

Compiler flags control individual suppression of optimization techniques, generation of floating point code (inline or for the 68881), formatting and contents of the listing and assembler output files, generation of source level debugger information, IEEE floating point operation modes, and register usage, among others. Reader service no. 24.
Sierra Systems
6728 Evergreen Ave.
Oakland, CA 94611
415-339-8200

PC Techniques, a new magazine for programmers, has been announced by **The Coriolis Group**. The first bimonthly issue will be published with a March/April 1990 cover date. The magazine will become a monthly publication in January of 1991.

PC Techniques will cover the DOS, Windows, OS/2, and Presentation Manager platforms. C, Pascal, Basic, and assembly language will be covered in every issue. Specialty languages like C++, Object Pascal, Smalltalk, and Actor will also find coverage.

The Coriolis Group was founded by *DDJ* columnist Jeff Duntemann and by Keith Weiskamp, occcasional *DDJ* author. *PC Techniques* is available for $21.95 for one year and $37.95 for two. Reader service no. 25.
Coriolis Group
3202 E. Greenway, Ste. 1307-302
Phoenix, AZ 85032
602-493-3070

Two new journals, *Inside Turbo C* and *Inside Turbo Pascal*, which offer programmers ongoing support of these two Borland languages, have been announced by **The Cobb Group**. The purpose of the two journals is to explore new algorithms, system tricks, and product updates, including complete source code. They will also contain tips, programming techniques, product news and reviews, as well as advice. And *Inside Turbo Pascal* covers OOP with Turbo Pascal.

Each journal costs $59 for 12 issues; sample issues are available. Source code in both issues can be downloaded from Cobb's BBS, for a yearly fee of $30. Reader service no. 26.
The Cobb Group
P.O. Box 24480
Louisville, KY 40224
800-223-8720

The original developer of Turbo Prolog, the **Prolog Development Center**

(continued on page 157)

(continued from page 152)
(PDC), has been granted the rights to the product by Borland International. The PDC will publish and market new versions under the name PDC Prolog. According to Michael Alexander at PDC, "The new version is a superset of the current Turbo Prolog. With the exception of the turtle graphics predicates, it is source-compatible with Turbo Prolog, so existing Turbo Prolog programs can be compiled 'as is' with PDC Prolog." And PDC Prolog supports the Borland BGI graphics interface.

A new DOS version should be available by now, and registered users of the DOS version of Turbo Prolog will be able to upgrade for $79. The OS/2 version should also be available, and will cost $599. Network support and a SCO 386 Unix version is scheduled for release in the second quarter of this year. Reader service no. 27.
Prolog Development Center
568 14th Street N.W.
Atlanta, GA 30318
404-873-1366

Intek C++ 2.0 is now available from **Intek Integration Technologies**. The company claims the product has as much power as AT&T's C++ 2.0 in an 80386 MS-DOS or Unix environment. Intek C++ 2.0 translates C++ code into C code. It supports most DOS C compilers, including Microsoft C, Turbo C, MetaWare High C and High C 386, Watcom C and Watcom C 386, and Novell Network C and Network C 386.

This support also includes the C extended keywords *near*, *far*, *huge*, *cdel*, *pascal*, and *fortran*, which makes it useful with Microsoft Windows and OS/2.

The Intek C++ translator uses 386 protected memory mode, and can compile large programs — up to 4 gigabytes. It supports multiple inheritance, type-safe linkage, new and delete operators as class members, overloading of the ->, ->*, and, operators, const and static member functions, and static initialization. It requires 1 Mbyte of memory, MS-DOS 3.1 or later or Unix System V/386, and costs $495. Reader service no. 28.
Intek
1400 112th Ave. SE, Ste. 202
Bellevue, WA 98004
206-455-9935

A C++ compiler for 80386/486 Unix-based systems has been released by **Peritus International**. In addition to AT&T C++ 2.0, the highly-optimized C++ compiler also provides support for K&R C and ANSI C; programmers select the appropriate C dialect by setting a compiler switch.

The compiler supports an extensive set of data types, including 8-, 16-, 32-, and 64-bit integers, IEEE-compatible 32-, 64-, and 80-bit floating point, user-defined aggregate types, and C++ class data types. The optimizations include global register allocation, constant propagation and folding, backward code motion with loop invariant removal, induction variable elimination, redundant store and dead code removal, and constant elevation.

Company president Ron Price told *DDJ* that Peritus intends on providing class libraries and development tools within the near future, including a package to provide a graphical interface to the X Windows system. He also said that the C++ is compliant to the AT&T 2.0 spec, except for multiple inheritance, which will also be supported in the near future.

The Peritus C++ compiler, which runs on 386/486 systems under SVR3 Unix and SunOS 4.0 Unix, sells for $1000. Reader service no. 29.
Peritus International
10201 Torre Ave., Ste. 295
Cupertino, CA 95014
408-725-0882

A few new assembly tools are now available. An assembly language library written entirely in assembly language has been released by **Quantasm Corporation**. Quantasm Power Lib (QPL) contains over 256 routines, provides high-level functionality, and has the ability to be customized.

QPL can be used by both novice and expert programmers. The documentation is coordinated with example programs on disk. The company claims that the compactness of QPL makes it convenient for programming memory resident programs or TSRs.

The product includes a menu and windowing system, over 75 string handling functions, extended precision math functions, a set of date/time functions, encryption/decryption algorithms, file name parsing, and sound control. The company intends to have high-level language interface routines available in the first quarter of this year. QPL requires MS- or PC-DOS 2.1 or above; 256K RAM; IBM PC/XT/AT, PS/2 or compatible; Microsoft MASM, Borland TASM, or SLR OPTASM. This product is not copy protected, nor has run-time royalties. The price is $99.95 without source code, $299.95 with. Reader service no. 30.
Quantasm Corporation
19855 Stevens Creek Blvd.
Cupertino, CA 95014
408-244-6826

From **Base Two Development** comes Spontaneous Assembly, an assembly-language library that contains over 600 functions and macros, including string and memory manipulation, near/far/relative heap management, doubleword/quadword integer math, date and time manipulation, and more. The company claims that every routine is hand-coded and optimized, and are easy to use because of the register-oriented parameter-passing convention. Company spokesman Alan Collins told *DDJ* that "this product does for 8088-family assembly language programming what Borland did for high-level language programming."

Spontaneous Assembly supports all Microsoft/Borland standard memory models, as well as custom models and mixed-model programming. The tool sports a full-overlapping windowing system with custom shadowing that allows direct memory via screen access or BIOS. DOS 2.0 or higher is required, and MASM 5.1 or TASM 1.0 are recommended. It costs $199, includes all source code, and comes with a money back, 60-day guarantee. Reader service no. 3.

Base Two Development
11 East 200 North
Orem, UT 84057
800-277-3625

Another is DASM, a disassembler for the 8086, 8088, and 80286, available from **JBSoftware**. DASM is able to disassemble and modify programs for which the source code is unavailable. It takes binary run files for DOS and compatible operating systems as input, and creates an assembly language file suitable for modification and reassembly as output. It acts as a virtual machine and maps the program being disassembled. It tracks register usage and determines the code, data, and labels, allowing the user to then edit the output and change the program.

DASM works by viewing commands and procedures in their real-time processing order, rather than in the sequence they appear in the program, which JBSoftware claims makes the programs easier to interpret and edit. Some of DASM's other features include the ability to generate appropriate ASSUMEs and segment maps, to handle multiple entry points, transfer vectors, and .EXE, .COM, and .BIN files up to 200K. It costs $250. Reader service no. 2.

JBSoftware
701 Cathedral St., Ste. 81
Baltimore, MD 21201
301-752-1348

Two new software products for Motorola's 88000 RISC microprocessor are available from **Diab Data**. The D-CC/88K, an optimizing C compiler, complies with the 88000 object code compatibility standard (OCS) and the Binary compatibility standard (BCS), and conforms to the proposed ANSI C standard. Optimizations include global common subexpression elimination, lifetime analysis (color), reaching analysis, automatic register allocation, loop invariant code motion, constant propagation and folding, dead code elimination, switch optimizations, and the ability to pass parameters into registers.

Diab's MC88000 toolkit is made up of the D-AS/88K Assembler, the D-LD/88K Linker, and the D-AR/88K Archiver. This package includes the D-CC/88K optimizing C compiler. The assembler is also OCS and BCS compliant, produces COFF object modules, supports standard MC88000 mnemonics, produces standard Unix directives for organizing code, among other things. The linker performs literal synthesis, generates warnings for unidentified external references, and is able to perform incremental links. The archiver maintains multiple files in a single archive file, and supports Unix System V command-line options. The compiler and toolkit are available for the Sun3/SunOS, Mac II/MPW, DECstation/Ultrix, and DEC VAX/VMS, among others. Reader service no. 33.

Diab Data Inc.
323 Vintage Park Dr.
Foster City, CA 94404
415-573-7562

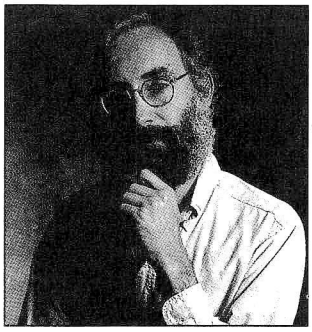
Books of Interest

A comprehensive treatment of concurrent programming techniques in the Strand programming language has been published by **Prentice Hall**. *Strand: New Concepts in Parallel Programming*, by Stephen Taylor and Ian Foster, covers an introduction to Strand, basic and advanced programming techniques, and how to apply Strand, with examples from both the academic and real worlds. The price is $30. ISBN 013-850587-X. Reader service no. 38.

Prentice Hall
Englewood Cliffs, NJ 07632
201-767-5937

DDJ

Pub Crawler



I read a lot of magazines. I read during meals, while talking to Jon on the phone, and while visiting the little programmer's room. I also follow magazine's fortunes, and I thought I'd pass along the latest rumors regarding some in which you may be interested.

CD-ROM End User. If you are interested in CD-ROM and haven't seen this, give it a look. Once you get past the uninspired name, the amateur editing, and the boring design, you'll find a bimonthly packed with information of solid value for both CD-ROM users and developers, written and compiled by knowledgeable people.

Embedded Systems Programming. Those whose realm is the other kind of ROM should know that *ESP* has gone to controlled circulation. What this means if you're a subscriber or potential ditto is that you may get it for free. What it means if you're an advertiser or potential ditto is that you can look for increased rates. It's a zero-sum game.

Micro Cornucopia. Dave Thompson is considering taking his 50-issue-old hacker's magazine monthly. He's looking for a "partner" — one with money to invest, I gather.

Microsoft Systems Journal. MSJ has been redesigned, and it's an improvement, though the publication still works too hard at being taken seriously. It's probably too much to expect that *MSJ*'s editors could learn from someone such as Dave Thompson how wit and playfulness can coexist with solid technical content.

Other captive magazines. Sun's user magazine is about to be sold — "given" is a better word, from what I hear of the deal — to IDG, publisher of *Computer World, InfoWorld, PC World, Macworld,* etc.; while Aldus has launched a magazine with a surprisingly drab look. The content is too self-serving, but the first issue contains a few good things, including what may be the most quick-and-dirty DTP how-to ever written, and an interview with Steve Ballmer on OS/2.

Ziff-Davis. The company that publishes *PC Magazine, MacUser, PC Computing, Digital Review,* and others (and that killed off *Creative Computing, Popular Electronics, PC Tech Journal,* and others) has been rumored for the past six months to be on the block. The rumors, which are making ulcers for Z-D employees, have been vehemently denied by Bill Ziff. The rumors are remarkably detailed: Pat McGovern, chairman of IDG, has perused the perspectus; Cahners, publisher of *Mini-Micro Systems,* has tendered an offer; the asking price is in the $800 million range; Goldman Sachs & Co. is handling the deal. If you believe Ziff's denials, you are led to believe that the rumors were started by one of Z-D's competitors. Whatever the truth, somebody is an awfully big liar.

Buzzwords

"Done deal" is one of those buzzwords that should buzz off, and I apologize for using it. Another buzzword that I hope won't catch on in the 90s is "experience," as in "user experience." Apparently the multimedia types within Apple are pushing to use it in the place of "user interface." I get the point, but I hope they keep this one in house.

My pick for the buzzword of the 90s is "facilitate." At least it has the right polysyllabic, academic aura. But I actually think it could be a GOOD buzzword. No, really. Here's why.

I believe fervently in the value of education, but I don't buy into the myth of teaching. The existence of this verb "teach" conveys the erroneous impression that it is possible to force-feed knowledge. The best teachers seem to understand that there is no such thing: Richard Feynman, on being given a teaching excellence award by the American Association of Physics Teachers, said, "I don't know how to teach. I have nothing to say about teaching," then went on to deliver a brilliant and entertaining lecture.

If you can't teach anyone anything, then all you can do is get out of the way, move any obvious obstacles aside, and let them learn. Facilitating learning, you might call it. The problem, I guess, is that it's hard to do. Clearing the student's path is one of those subtle acts that succeeds only by making itself invisible.

Like good writing, and like good user interface design. Good writer Esther Dyson discussed the desktop metaphor in the January issue of *PC Computing,* saying that it "is not meant to suggest that the computer is a desktop, but to provide a sense of recognition and reasonable expectations. This metaphor, so popular now, suggests tasks the computer can reasonably be expected to do." Suggest things. Create an environment the user can explore, letting the user discover things by recognizing the familiar and following reasonable expectations into the unfamiliar. Get out of the user's way. Facilitate. Yeah. I like the word. The trouble is that if it catches on, people will start ringing the changes on it: facilitator, facilitation, facile. And sooner or later some user is going to walk into a computer store and ask to be shown the facilities. And be taken to the little programmer's room. Might be all right if there are some good magazines in there.

Michael Swaine

Michael Swaine
editor-at-large

#163 APRIL 1990 $2.95 ($3.95 CANADA)

15 YEARS OF POWER PROGRAMMING

Dr. Dobb's JOURNAL

SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

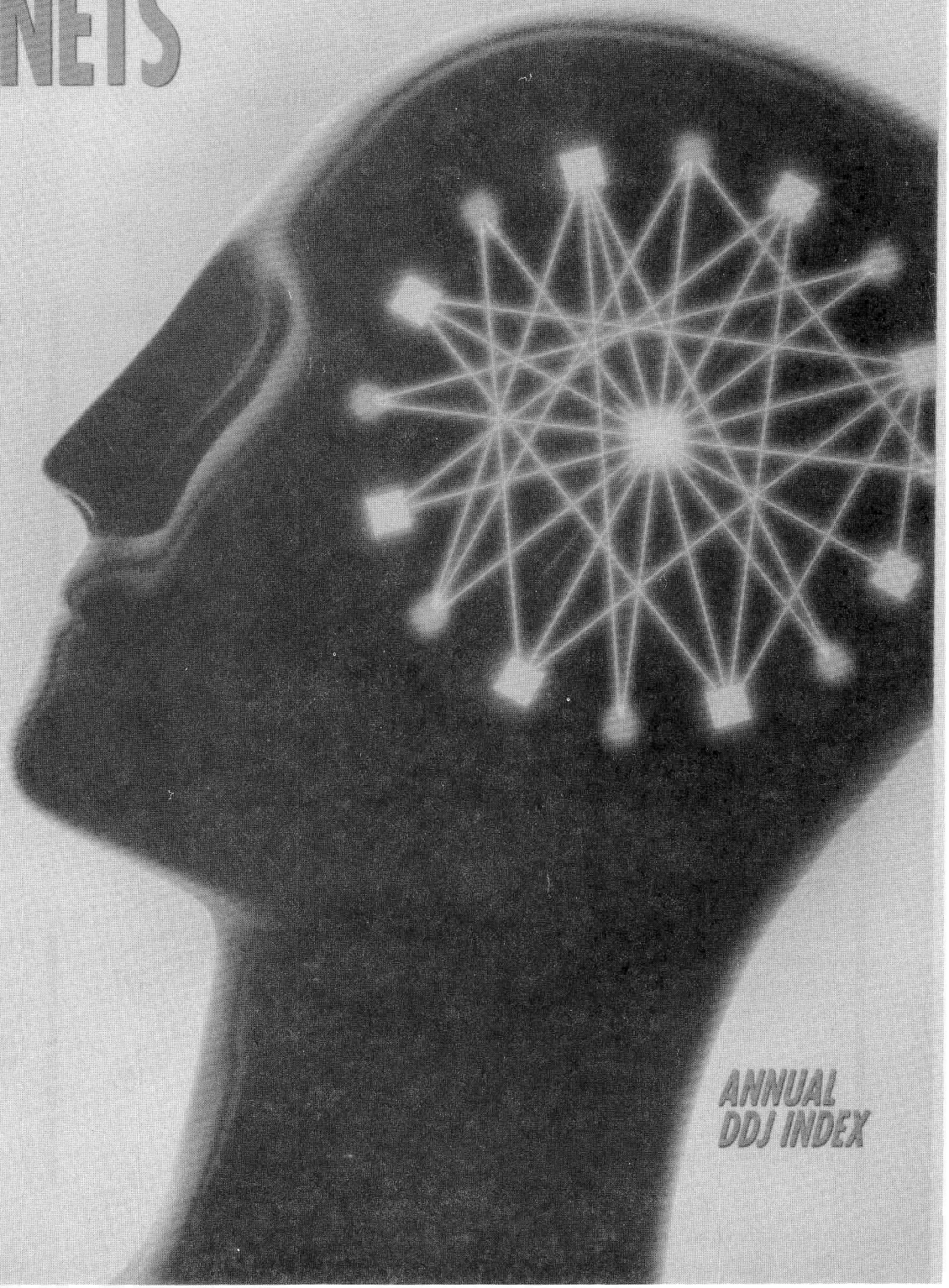
NEURAL NETS NOW!

- BAM IN C++
- MODULARIZING NEURAL NETS
- IMAGE PROCESSING

VESA VGA

IMPLEMENTING RHEALSTONE

DATA COMPRESSION



ANNUAL DDJ INDEX

0 38351 16562 8 04

CONTENTS

APRIL 1990
VOLUME 15, ISSUE 4

FEATURES

BIDIRECTIONAL ASSOCIATIVE MEMORY SYSTEMS IN C++ **16**
by Adam Blum
Bidirectional associative memory is a neural net model that may solve the content-addressability problem. Adam implements BAM systems using C++, discovering that object-oriented languages go hand-in-glove with neural net development.

A NEURAL NETWORK INSTANTIATION ENVIRONMENT **28**
by Andrew J. Czuchry, Jr.
Developing useful and efficient network architectures requires a simple, yet flexible environment. Andy presents an environment that dynamically creates neural networks.

UNTANGLING NEURAL NETS **38**
by Jeannette "Jet" Lawrence
With more than 40 functioning neural net models to choose from, it is important to understand their similarities and differences.

IMPLEMENTING THE RHEALSTONE REAL-TIME BENCHMARK **46**
by Rabindra P. Kar
It's been over a year since *DDJ* first introduced the Rhealstone, a set of benchmarking operations for real-time multitasking systems. Robin presents the "refined" definition, along with a suite of C programs to implement the benchmark.

BOUNDING BOX DATA COMPRESSION **56**
by Glenn Searfoss
The "bounding box" method of data compression is fast and efficient for bit-mapped data. Glenn describes this technique and compares it to the better-known RLE.

1989 *DDJ* INDEX **65A**
compiled *by Janna Custer*
Here's an index to articles and subjects in last year's *DDJ*.

VESA VGA BIOS EXTENSIONS **65H**
by Bo Ericcson
The VESA VGA BIOS extensions make it possible to write generic graphics software that tap into the powerful capabilities of Super VGA.

EXAMINING ROOM

CRUISING WITH TOPSPEED **72**
by Alex Lane
TopSpeed C launches JPI into the C arena. Alex puts the package under the Doctor's microscope, paying special attention to the C TechKit, and has some fun in the process.

PROGRAMMER'S WORKBENCH

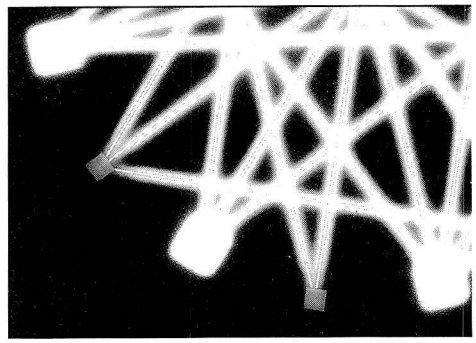
NEURAL NETWORKS AND IMAGE PROCESSING **77**
by Casimir C. "Casey" Klimasauskas
Casey explores different approaches to edge enhancement systems, first using C, then using an off-the-shelf, two-dimensional array engine called Lotus 1-2-3.

COLUMNS

PROGRAMMING PARADIGMS **119**
by Michael Swaine
SD '90 brought a little of everything, from serious talk about new paradigms to hype about new products. Here's Mike's report on the year's most important programming conference.

C PROGRAMMING **127**
by Al Stevens
Al starts to sort things out with CSORT, a sorting facility that you can use from within your programs or from the command line.

STRUCTURED PROGRAMMING **135**
by Jeff Duntemann
The time has come for a time-and-date stamp object. Jeff develops one using Turbo Pascal, after mulling over Xerox's suit against Apple.



DEPARTMENTS

EDITORIAL **6**
by Jonathan Erickson

LETTERS **8**
by you

SWAINE'S FLAMES **160**
by Michael Swaine

PROGRAMMER'S SERVICES

OF INTEREST **152**
compiled *by Janna Custer*

ADVERTISER INDEX **153**
where to go for more information on products

PROGRAMMER'S MARKETPLACE **154**
classified ads

NEXT ISSUE

Bigger programs can lead to bigger memory management headaches. In May, we'll examine different approaches to memory management and continue our in-depth coverage of 80386 programming.

It Takes More Than Nerve

The biggest fear of those who champion neural networks is guilt-by-comparison with the artificial intelligence camp. They're not alone in this. Object-oriented advocates, as well as most other popular technologies that make the front pages of pseudo-technology news tabloids, don't want to be snake-bit by the same type of hype that poisoned AI development. The frontal assaults of AI and expert systems, fueled by big money and bigger promises, have been nonexistent for neural nets and, although those neural net developers trying to eke out a living might disagree, the lack of venture capital has probably been a blessing. Less hype buys more time, at least as long as enough money comes in to keep the lights turned on.

Nevertheless, there continues to be a lot of interest and development in neural nets. A survey recently published by Future Technology Surveys of Madison, Georgia listed over 200 companies and organizations currently producing neural-related products or undertaking serious neural net research. And it just isn't the little guys doing all this research, either.

Among the big outfits testing the neural net waters is Intel. Early last summer, a ten-member Intel engineering team, under the direction of Mark Holler, rolled out an Electronic Trainable Artificial Neural Network (ETANN) chip that is capable of up to 2 billion multiplies and accumulates per second. To put the chip to work, Holler and his crew have built a prototype ETANN-based board that plugs into the PC AT bus; Mark Lawrence and the folks at California Scientific (developers of BrainMaker, a neural net simulation package for PCs) are developing the software tools that let you use the system. There's also a rumored Intel research project that will put a version of the ETANN board into an i860-based system that can achieve 33 billion connections per second.

Intel isn't the only big IC manufacturer poking around in neural nets. A few months ago, Sharp introduced a neural-network image-processor chipset that simulates human vision and, the company claims, supports PC applications at speeds up to 700 MIPS.

These examples illustrate another trend in the neural net world — a transition from software to hardware. Within ten years, or so say the experts, more neural nets will be implemented in hardware than software. Until then, engineers will begin to overcome many challenges, including the implementation of back propagation in hardware and the parallelization of the entire scheme.

So where does this leave software developers? For one thing, a whole new class of development tools is in the offing, designed for specific neural net hardware implementations. Another type of tool will be like that described by Andy Czuchry in this issue, whereby designers can match the right neural model with the task at hand. Nor will the simulators go away; they may be used to simulate the right net with appropriate learning, then generate source code to be frozen in silicon.

In her keynote address at Miller-Freeman's SD'90, Smalltalk pioneer and ParcPlace System's president Adele Goldberg expressed a concern similar to that I wrote about in this space last month — the spread of litigation and its effect on the software industry.

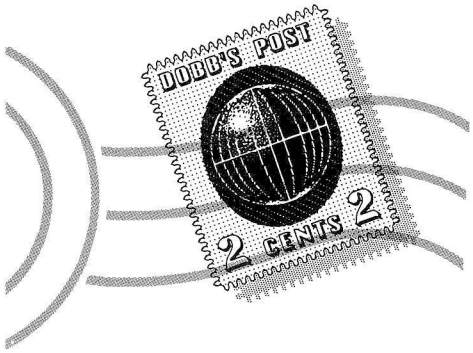
Although her talk concerned a wide variety of legal issues — from intellectual property to the emerging problem of who owns the design and implementation of objects, as in object-oriented programming — she spent a fair amount of time on copyrights and patents. "Lawyers will always tell you two things," she said, "try to patent or copyright whatever you do." She went on to describe a speech she gave to a group of lawyers, where she was asked how to convince software developers to protect their works. "My answer was simple," she said. "Tell them to protect their work so that they have the choice later on to give it away." Not doing so, she explained, opens the door for someone to come along and take it away. But Goldberg wasn't engaged in lawyer-bashing, no matter how easy that is. What she was presenting was a persuasive argument for open standards and open licensing. She pointed out that among the problems litigation forces upon us are the waste of time and money, the fear of alliances, the inability of entrepreneurs who lack clear patent or copyright protection to attract investors, and the expense of starting up new businesses.

One of the main points of her talk was simply to "reassert an often unstated goal of our industry — to share ideas and to challenge one another with our innovative expressions of those ideas." Nicely put.

And no, the favored horse running in the first race at Bay Meadows racetrack the other night wasn't our official mascot, even though the nag's name was "Dr. Dobbs." Although, he lost by a nose in a photo finish, the good Doctor is surely chomping at the bit to get into the next race. We'll keep you posted on his progress this season.



Jonathan Erickson
editor-in-chief



Faster Animation

Dear *DDJ*,

I enjoyed Rahner James's article on "Real Time Animation" in the January 1990 issue of *DDJ*. Ever since the price of EGA devices dropped, magazines have been filled with how-to articles, but few have covered icons or sprites. In 1986 I ported a collection of graphics routines from my Zenith 100 system to the EGA. These routines are now part of an icon development environment called "ProGraphx Toolbox," available from Stanwood Associates.

Speed is definitely the key to success in graphics, and Mr. James's routines definitely are fast. In my routines I have come up with another approach, which, I believe, is slightly faster in displaying icons. Since the EGA is latched, you must determine which planes will be accessed during a write, a clock cycle consuming process. Mr. James's routines store one byte per pixel, enabling 128 colors and an intensity bit. If the format of the sprite were laid out plane by plane rather than pixel by pixel, the function could set the registers for a plane and then write all the data for that plane. The function would continue plane by plane, only setting up the register once per plane per icon. Additionally, the EGA does not allow bit access. So why not place a byte's worth of data on the screen during each write rather than only one new bit? I enjoy seeing quality articles every time I open a new issue of *DDJ*. Keep it up!

Peder Jungck, Stanwood Assoc.
Chicago, Illinois

On Location

Dear *DDJ*,

The excellent article "Location is Everything," by Mark Nelson, (January, 1990) was most timely. Once again *DDJ* came up with just what I wanted just when I needed it. I think, however, there is a small problem with the code.

Mark uses the exe header field "Displacement of stack in Paras" (0e) to locate the start of the initialized data area. This works only when the amount of initialized data is less than one paragraph long since the stack displacement corresponds to the end of the initialized data area. In the general case, the program needs the starting paragraph. I was able to easily find this value by parsing it out of the .MAP file. Using this value in the relocation function causes the program to perform as advertised.

```
/* input_base_data_segment is a global
                                unsigned int */
/* data_seg is declared as char
                                data_seg[81] */
/* map_file is a FILE * to the .MAP file
                                created by */
/* Scan the .map file for the word "BSS"
                                */
while(strcmp(data_seg, "BSS") ! = 0)
      fscanf(map_file, "%s", data_seg);
/* The next field in the file is the loca-
                                tion of the */
/* data. */
fscanf(map_file, "%s", data_seg);
data_seg[strlen(data_seg)-1] = '\0';
/*kill the 'H' */
input_base_data_segment =
                        htoi(data_seg) >> 4;
```

In the function *process_relocation_table( )*, replace all references to the variable *first_data_segment_in_exe_file* with *input_base_data_segment*.

Stephen J. Beaver
Winchester, Virginia

Mark responds: As Stephen Beaver notes, there is a field in the header portion of an EXE file that tells me where the start of the program stack segment is located. I use this field to determine where RAM data starts in the EXE file. Mr. Beaver must have taken note of the lines in my START.ASM file shown below. Because the stack segment follows the DATA, BSS, and CONST segments, Mr. Beaver concludes that the value I calculate for the start of RAM actually points past all these segments. However, there is an additional segment definition line a little farther down in the START.ASM file:

```
DGROUP    GROUP _CONST, _BSS,
                        _DATA, _STACK
```

This statement causes the compiler and linker to gather all four of these segments together into one segment. This means that all four segments are collected together, and the pointer to the start of the stack segment will actually point to the start of DGROUP, which will be at the start of the _CONST segment. So, if you use the START.ASM startup file as I required, the LOCATE.EXE program will work properly. By the way, Mr. Beaver "solved" this problem by modifying my LOCATE program to read in a MAP file that the linker has produced. Reading in MAP files to drive a Locate program is not a bad approach. In fact, Jensen & Partners International are providing a TSLOCATE utility with their new TopSpeed C compiler, which does just that. I chose to avoid this approach for a couple of reasons. First, there is no standard MAP file format. Every linker is free to create their own format, and a good LOCATE program would be forced to continually adapt to these. Second, a program using this approach is vulnerable to errors caused when the MAP file is not actually the one from the latest link. Because the information I wanted was in the EXE file, and the EXE format is standardized across all MS-DOS compilers, I elected to not read in the MAP file. *I hope this clears up some of the confusion. Dealing with program segments at a low level is usually concealed from HLL programmers by the compiler, for which we can all give thanks.*

Random Structures

Dear *DDJ*,

This is in response to the December 1989 letter by Dan W. Crockett. He is treating the term "structured" and the term "modular" as being equivalent. A structured module, program, or system

(continued on page 12)

```
_TEXT       SEGMENT BYTE PUBLIC 'CODE'
_TEXT       ENDS
END_OF_ROM  SEGMENT PARA PUBLIC 'STARTUP_CODE'
END_OF_ROM  ENDS
_CONST      SEGMENT PARA PUBLIC 'CONST'
_CONST      ENDS
_BSS        SEGMENT WORD PUBLIC 'BSS'
_BSS        ENDS
_DATA       SEGMENT WORD PUBLIC 'DATA'
_DATA       ENDS
_STACK      SEGMENT WORD STACK 'STACK
MYSTACK     DB 512 DUP (?)
_STACK      ENDS
```

(continued from page 8)
need not be modular, and a module may or may not be structured. Also, a module, program, or system that has "spaghetti code" has a structure. It is called a random structure.

One of the purposes of using modules is that the code is reusable. We use this all the time. The modules are in libraries. Examples are the Fortran library routine SIN(x), the Cobol COPYLIB file, and the C library routine sin(x). The proper term for module tree relationship is a "caller" module and a "called" module. This describes the relationship much better than the "father" – "son" model. Further, a module should not know anything about the "caller" module, other than what is in its argument list. Can you imagine the chaos that would result if we had to rewrite SIN(x) or sin(x) every time they had more than one caller?

Ned Logan
Seattle, Washington

Pascal Participation, Pleeez

Dear *DDJ*,

After reading Terry Ritter's letter entitled "Standardizing the Standardizing Process" in your February 1990 "Letters" column, I have the feeling that many readers may not understand the standardization process and will be given a false impression.

Membership on X3J9, the other X3 committees, and the IEEE committees is not restricted to some elitist group. Membership is open to all interested parties who are willing to participate. Users are especially encouraged to participate.

Decisions are not made in a back room behind closed doors. Committee meetings are open to the public with visitors and observers not only welcome but encouraged. Likewise, committee documents are open to the public and people who cannot attend meetings can become official observers and receive all committee mailings.

Consensus is the method by which most decisions are made both at the international level and at the domestic level for Pascal. However, it is the consensus of those who participate.

It was not only the consensus, but the unanimous vote of both the International Working Group on Pascal and the American National Standards Committee on Pascal that no action be taken on some of Mr. Ritter's comments for the Extended Pascal standard. The main reason for this was that major changes and development would have been required and it was felt that this should be handled separately rather than unduly delay the standard, which was in its final stages of review.

This does not mean that his comments have been shoved under the table. Many of the areas brought up by his comments, including exception handling, alphanumeric labels, and multiple arithmetic data types, are being worked on by the committee. The intent is to issue information bulletins, technical reports, and addenda to standards when work on them is completed. User participation is encouraged in this work, especially now when it is still in a nearly stage of development.

The committee is also now beginning to look at object-oriented extensions to Pascal, and has submitted a project proposal to its parent bodies for approval of this work. It is expected that approval will be received early this year. When this approval is received, announcements will be submitted to all publications (including *Dr. Dobb's* and similar user-oriented publications) that might have an interested audience.

People from Apple, Borland, Microsoft, and other vendors are planning to participating in the object work. User's views, and user participation, at this early stage would be especially welcome.

I encourage Mr. Ritter and other users to participate in Pascal and other standardization efforts. No one needs to elect you. All you need to do is participate.

For users that do not have financial support, there are organizations (such as SIGPLAN) that have funds allocated for this purpose.

To find out more about participating in the Pascal standards activity please contact me by letter, phone, FAX, or e-mail.

Thomas N. Turba
Chairman X3J9, Pascal
Unisys Corp., MS: WE3C
P.O. Box 64942
St. Paul, MN 55164
612-635-2349; 612-635-2003 (Fax)
NET: turba@rsvl.unisys.
comuunet!s5000!turba

There's More than One Way to Get From Pascal to C, or Return of the Living Fugu-Eaters

Dear *DDJ*,

I too enjoy Jeff Duntemann's writings, though not for the same reason as does Dale Lucas ("Letters," Jan. 1990). I love a good argument. So I get a kick out of reading Jeff's ravings against C, all the while thinking up incontrovertible (I'm sure they *must* be) refutations.

I do agree with him that C is ugly. It looks like Dagwood's dialogue after he hammers his thumb, %*!&( )*#$@! But I put up with that for the sake of the language's abilities. Jeff, on the other hand, sees no redeeming value in C, whatsoever, as we were so forcefully reminded in the January issue.

Recall that Dale Lucas asked him whether there's a way to call a third-party (*sans* source) C library's routines from a Turbo Pascal program. This little spark lit the fuse to one of Jeff's best tirades to date, in which he accused C programmers of acting macho, of neglecting to neck with their spouses and play with their dogs, and (this was the killer blow) of EATING FUGU!

Oh boy, did he give it to us. Unfortunately, he got so carried away with his ranting twaddle that he neglected to help his Pascal co-linguist. He told Dale to rewrite the whole library in Pascal!

If you don't mind taking advice from a fugu-eater, Dale, I think there's a way to hook those C routines. But first a question: Doesn't Turbo Pascal have something akin to Microsoft's "[C]" attribute, which you append to a procedure declaration to tell the compiler to use C's calling and naming conventions? Guess not, or the problem would be trivial and you wouldn't have written.

So you'll need to turn to a more powerful language — ummmh, let's say C — to write the hooks. Your third-party library will have given you a header file, for instance "WINDOW.H," declaring its functions. For example:

```
int WinCreate(int height, int width, int
                                    color);
void WinOpen(int winnumber, int
                        xcord, int ycord);
```

And so forth. Add to this file a new Pascal-callable hook function for each declaration, thusly:

```
int pascal HOOK_WinCreate(int height,
                    int width, int color) {
return( WinCreate(height, width, color
                                    )); }
void pascal HOOK_WinOpen(int win-
number, int xcord, int ycord.)
{WinOpen(winnumber, xcord, ycord);
                                    return;}
```

Rename the file, say to "HOOK.C," and compile it. Finally, translate the hook function prototypes into declarations for your Pascal modules:

```
function HOOK_WinCreate
(height,width,color : integer) : integer;
                                    extern;
procedure HOOK_WinOpen (winnum-
    ber, xcord,ycord : integer) :extern;
```

(Do I have that right? I read Pascal but speak it poorly.) The C code above works on my Watcom compiler, and

(continued on page 14)

(continued from page 12)
ought to work on QuickC as well. Of course, I've begged the more difficult questions like memory models and translating C strings and structures into Pascal. But this might be enough to get you started.

A couple of closing questions. Dale . . . wouldn't it be easier just to code your app in a real-man's language like C? And Jeff . . . what the hell IS fugu, anyhow?

Bob Twilling
Bozeman, Montana

Dear *DDJ*,
In your January issue, you printed a letter from Dale Lucas asking for help interfacing Turbo Pascal to C. Jeff Duntemann's response spent more effort bashing C than helping Mr. Lucas. I'm not particularly fond of C either, but I think I have a very simple solution.

I know nothing about Turbo Pascal, but if it uses (or can be made to use) the same calling convention as Microsoft Pascal, we're in luck. Simply use Microsoft QuickC to create one-line helper functions that translate the calling conventions. These helper functions would be declared as "pascal" functions, and thus be callable directly by Pascal. The only statement in the function would be a call to the library function using the C calling convention. For example:

```
int cdecl foobarC(int,int); /* This is in
                                the library */
int pascal foobarPascal(int arg1, int arg2)
{
return ( foobarC(arg1, arg2) );
}
```

This assumes the C-based library has a function called *foobarC( )*, which has two integer arguments and an integer return value. The function *foobarPascal( )* passes the arguments in the return value out.

Tim Paterson
Renton, Washington

Dear *DDJ*,
This letter is in response to Jeff Duntemann's answer to Dale Lucas's letter in the January 1990 issue of *Dr. Dobb's Journal.*

I disagree with Jeff's answer. A Pascal routine can call a C routine by using an impedance matching routine written in assembly. The routine takes the Pascal arguments, pushes them on the stack, calls the C routine, cleans up the arguments pushed, and then cleans up the stack for the Pascal caller. A macro can be built, which has the Pascal entry point, the matching C entry point, and the size of the arguments in bytes. The macro *CHook* in Listing One (below) implements this.

The Pascal programmer simply calls the Pascal entry point. The impedance matcher handles the language differences and returns. Simple, easy, and direct. Much better than recoding a debugged, commercial library.

By placing the code in a macro, the user can just build a table of macro calls which reflect all entries to the C library. Each macro expands and builds the impedance matching code for each library entry point.

Some notes involving the use of the macro:

- The argument size is given in bytes, not number of arguments. You must determine the number of bytes by adding the number of bytes in each argument that is passed.
- This impedance matcher will work for Pascal procedures. For functions, you will have to make sure that your flavor of Pascal uses AX for 16-bit return values and DX:AX for 32-bit return values. If you need to map the function return values, just add this to the macro.
- I coded this macro for large model programs. Small model programs must adjust the value added to SI from 6 to 4.
- If SS matches DS at all times, the push/load/pop of DS can be removed.
- The argument transfer can be sped up by using a MOVSW and dividing the count stored in CX by 2. This should always work because C requires the minimum argument by an int (2 bytes under MSC).
- If the size of the arguments is 0, the argument transfer code can be eliminated using conditional assembly.
- Normally, the name of a C routine starts with an underscore. This could be included in the macro instead of requiring an underscore for every *CHook* invocation.
- The impedance match does take time. If you have a very time-critical call, you may have to recode the routine in Pascal or directly in assembly. However, the macro can get you up and running quickly.

This code has not been tested with Turbo Pascal. It has only been tested using a Microsoft C program (Listing One) that calls the Pascal entry using the pascal keyword. If the macro doesn't work with Turbo Pascal, it should only take a small amount of tweaking to make it work. The key is to draw a picture of your stack frame and test it in Debug, following the argument flow.

Jim Shimandle, Primary Syncretics
Santa Clara, Calif.

DDJ

Listing One

```
;
; c2pas.asm
; C/PASCAL impedence matching module
;
        .model  large
code    segment para public 'code'

;-------------------------------------------------------------------

CHook   MACRO   PascalEntry, CEntry, ArgSize
        EXTRN   CEntry:FAR
        PUBLIC  PascalEntry
PascalEntry     PROC

        PUSH    BP              ; Make stack frame
        MOV     BP, SP

        PUSH    CX              ; Save registers
        PUSH    SI
        PUSH    DI
        PUSH    DS

        MOV     CX, SS          ; Set DS to point to stack
        MOV     DS, CX

        SUB     SP, ArgSize     ; Save space for arguments
        MOV     CX, ArgSize     ; Set count for arg transfer
        MOV     SI, BP          ; Get frame pointer
        ADD     SI, 6           ; Point to start of PASCAL arguments
        MOV     DI, SP          ; Point to start of C arguments
        CLD                     ; Move is up
        REP MOVSB               ; Move the arguments

        CALL    CEntry          ; Call the C routine
        ADD     SP, ArgSize     ; Remove arguments from stack

        POP     DS              ; Restore registers
        POP     DI
        POP     SI
        POP     CX

        POP     BP              ; Restore frame
        RET     ArgSize         ; Exit

PascalEntry     ENDP
        ENDM

;-------------------------------------------------------------------

; Invoke macro for test routine

CHook   p_sum3, _c_sum3, 6      ; Invoke macro for:
                                ;   p_sum3 is PASCAL call
                                ;   c_sum3 is C library routine
                                ;   6 is the number of argument bytes

;-------------------------------------------------------------------

code    ends
        end
;
; end of c2pas.asm
;
```

End Listing

Bidirectional Associative Memory Systems in C++

Recent innovation makes associative memory practical for real-world problems

Adam Blum

Content-addressability was always a goal of early neural network pioneers. It is a quest that has been pursued by computer scientists in general for decades. However, the goal has proved highly elusive. Search time has always depended on the amount of data stored, although much research has gone into reducing the slope of this curve. Real-time pattern recognition (as applied to any number of fields, be it speech recognition, radar signature identification, or part classification) is still far from reality. One particular neural-network construct, bidirectional associative memory (or BAM), has promised some solution to this problem.

I'll first describe the BAM concept, then show you how a relatively recent construct, the Bam System, can make it immediately feasible for real problems. Finally, I'll present an actual implementation of the Bam System written in C++.

As developed by Bart Kosko, BAMs are a neural-network-based attempt at content-addressable memories. They are based on a two-layer feedback neural network. They attempt to encode m pattern pairs (A_i,B_i) where $A_i \in \{-1,+1\}^n$ and $B_i \in \{-1,+1\}^p$ in an $n \times p$ matrix M. BAMs are globally stable and provide instant recall of either of the two-pattern pair elements. However, BAMs face some limitations. For large pattern lengths, n, storage requirements increase $O(n^2)$. More importantly, storage capacity is only, on an average, $m < \min(n,p)$. Thus, for moderate pattern lengths, capacity of the matrix M becomes a problem. Recent research promises help for this problem. However, some initial description of BAMs should be made.

Adam is a programmer analyst at Ketron Inc. of Arlington, Virginia, and is the principal developer of several commercial software packages. His interests include compiler design, C++, and (of course) applications of neural nets. He can be contacted at 1700 N. Moore St., Ste. 1710, Arlington, VA 22209, or on CompuServe at 72650,1773.

Encoding

BAM encoding is accomplished by simply summing the correlation matrices of each of the pattern pairs. That is, the matrix that encodes the first m pattern pairs, M, is simply:

$$M = \sum_{i=1}^{m} A_i^T B_i$$

Thus, to encode a pattern pair, simply produce its correlation matrix, $A_i^T B_i$, and add the values to the current matrix M. For discrete implementations, it so happens that the matrix arithmetic works out better if 0s and 1s are encoded as -1s and +1s. So the first step in the process will be to convert any {0,1} string to {-1,+1}. Example 1 shows this process.

Note that we can erase association (A_i,B_i) from M by adding $-X_iTY_i$ to M. But if we are using a {-1,+1} representation, this is the same as adding (A_i,B_iC) or (A_iC,B_i) to M (where C represents the pattern's complement). This fact will become important in our implementation of the BAM system.

Decoding

After we have "trained" our BAM with the m pattern pairs (A_i,B_i), we want the BAM to recall pattern B_i every time A_i is presented to the matrix (and, conversely, recall A_i every time B_i is presented to the matrix). It turns out that BAMs also have the property that B_i will be recalled every time something close to A_i is presented. Example 2 outlines the steps involved in the decoding process.

But it won't go on forever. As shown in Example 2, eventually the fields will "resonate" to steady patterns. This property of BAMs is called "global stability." Lyapunov energy functions allow us to prove that BAMs are globally stable.

Energy Functions and Stability

Lyapunov showed that any function expressed in terms of the system parameters that is zero at the origin and has nonincreasing changes is globally stable.

An energy function for the BAM can be $E(A,B)= -AWB^T$. This function is obviously zero at the origin (that is, zero when A and B are zero). We just need to show that it has nonincreasing changes. Well, $\Delta E_A(A,B)= -AWB^T$ and by the definition of our function *f*, each A_i in A will be positive only if W_iB is positive. If A_i is negative, W_iB must also be negative. Thus the change in energy will always be negative or zero. The system is thus globally stable.

Adaptive BAM

As we have just described it, the connection matrix M is simply the sum of the correlation matrices of the patterns presented to it. We can use more sophisticated equations to allow faster convergence or more accurate recall. As long as such equations can also be shown to converge, we should have no problem with this.

The simplest of these learning laws is called Hebb's law: $m_{ij} = -m_{ij} + f_i(x_i) * f_j(y_j)$, where m_{ij} is the connection weight between the neuron x_i and neuron y_j, and f_i and f_j are the threshold activation functions for x and y, respectively.

Other laws that could be used include competitive learning and differential Hebb; there is much research on which of these is most effective. In our implementation, we will be presenting a simple nonadaptive BAM. However, it is easily extensible to the learning function of choice.

Problems

BAM faces two problems, the first of which is that the amount of storage taken up varies $O(n^2)$, where n is the pattern length (actually, it will vary O(np) where n is pattern length of A and p is pattern length of B).

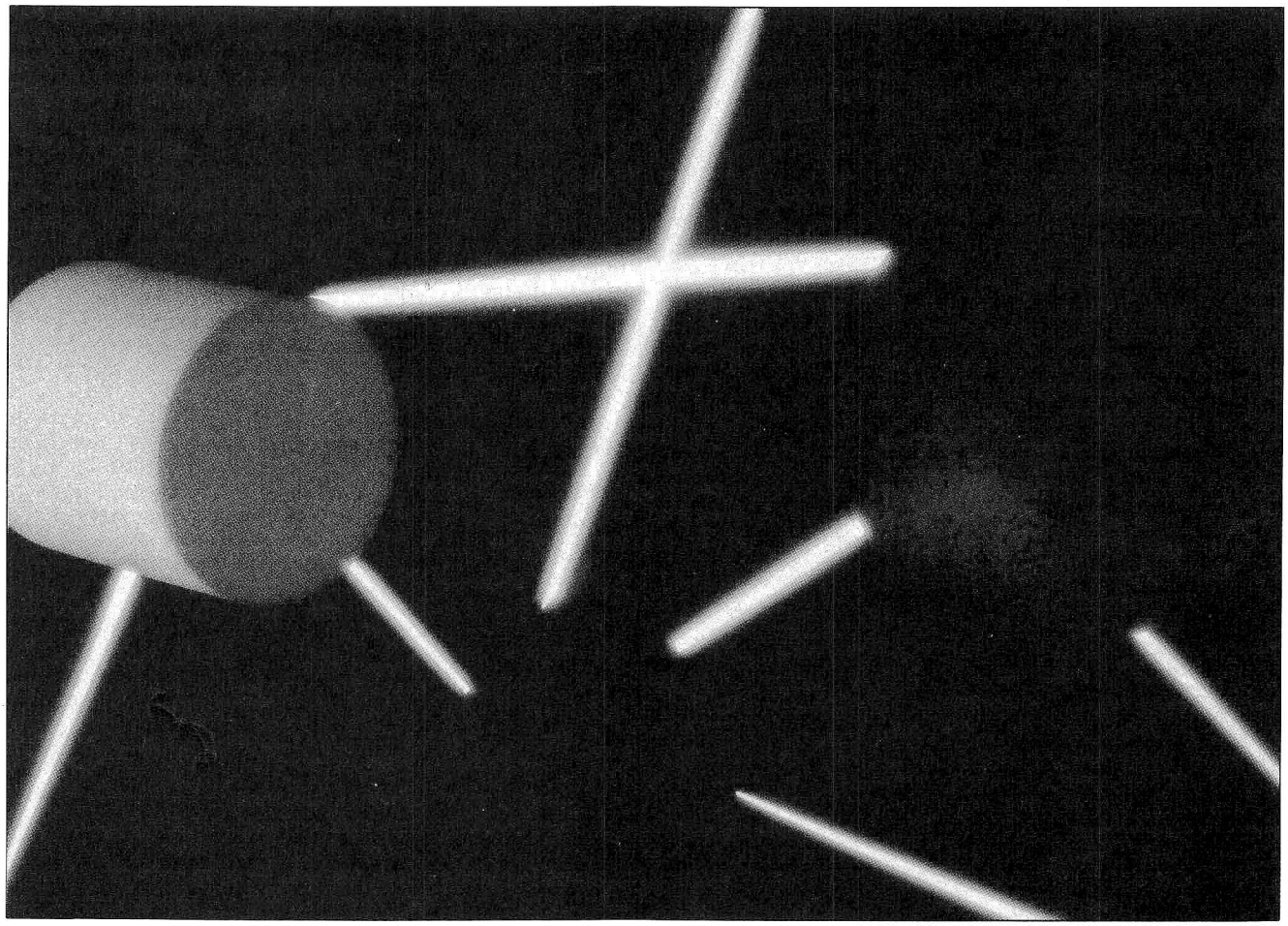
The second problem — capacity — is more critical. Reliable retrieval of associations begins to degrade when the number of patterns stored, m, is greater than the minimum of the two-pattern dimensions. In other words, to be reliable the matrix capacity is $m < \min(n,p)$.

For large pattern lengths, this is not so much of a problem, but many applications have inherently moderate pattern lengths. We intuitively find it almost obvious that if a BAM can store only up to the minimum of its pattern lengths, it will be virtually useless for real-world applications.

BAM Systems

In 1989, Patrick Simpson of General Dynamics published a paper introducing the concept of a "BAM System." This is a rather uninformative name for a system that allows for multiple matrices when one matrix's capacity is saturated. Perhaps a better name would be "Multi-Matrix BAM" or, because each matrix is just a representation of the connectivity between the two patterns, "Multi-Connective BAM." Anyway, it is an inventive way to overcome the severe problem of matrix capacity.

The Bam System operates as follows: Pattern pairs are encoded one by one in a single BAM matrix, M_1. After each pattern pair is encoded, the matrix must be tested to ensure that each pattern pair stored can be recalled. If a pattern pair cannot be recalled, the current pair is removed from the matrix. We then attempt to store the pair in another connection matrix. We continue to try to store it in other matrices, M_i, until it is stored such that all pattern pairs in that matrix



Dr. Dobb's JOURNAL
SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

PUBLISHER *Peter Hutchinson*

EDITORIAL
EDITOR-IN-CHIEF *Jonathan Erickson*
MANAGING EDITOR *Monica E. Berg*
TECHNICAL EDITORS *Michael Floyd, Ray Valdes*
EDITORIAL ASSISTANT *Janna Custer*
CONTRIBUTING EDITORS *Al Stevens, Jeff Duntemann, Martin Tracy, David Betz, Tom Genereaux, Andrew Schulman*
COPY EDITORS *Rhoda Simmons, Pamela Dillehay, Nan Fornal*
EDITOR-AT-LARGE *Michael Swaine*

ART/PRODUCTION
ART/PRODUCTION DIRECTOR *Larry L. Clay*
ART DIRECTOR *Michael Hollister*
PRODUCTION SUPERVISOR *Amy Shulman Lesovoy*
TYPOGRAPHERS *Teresa Raines, Margaret Anderson, Charlene Carpentier*
COVER PHOTOGRAPHER *Michael Carr*

CIRCULATION
DIRECTOR OF CIRCULATION *Maureen Kaminski*
CIRCULATION MANAGER *Randy Robertson*
CIRCULATION PLANNING MANAGER *Manny Sawit*
DIRECT MARKETING MANAGER *Andrea Weingart*
NEWSSTAND MANAGER *Sarah Forsman*
DIRECT MARKETING COORDINATOR *Francesca Davies*
PROMOTION COORDINATOR *Pam Moore*
FULFILLMENT COORDINATOR *Anne Jean*

ADMINISTRATION
VICE PRESIDENT OF FINANCE *Kate Deschamps*
CONTROLLER *Mary Collopy*
CREDIT MANAGER *Betty Arsene*
ACCOUNTING SUPERVISOR *Renate Kernke*
ACCOUNTS RECEIVABLE *Wendy Ho*
ACCOUNTS PAYABLE *LuAnn Rocklewitz*

MARKETING/ADVERTISING
DIRECTOR OF SALES AND MARKETING *Karla Spormann*
ADVERTISING COORDINATOR *Laura Stack Pullen*
MARKETING ASSISTANT *Sara Noah Ruddy*
ACCOUNT MANAGERS *see page 153*

M&T PUBLISHING INC.
CHAIRMAN OF THE BOARD *Otmar Weber*
DIRECTOR *C. F. von Quadt*
PRESIDENT *Laird Foshay*
VICE PRESIDENT OF PUBLISHING *William P. Howard*
VICE PRESIDENT/GROUP PUBLISHER *Randall L. Stickrod*

DR. DOBB'S JOURNAL (USPS 307690) is published monthly, except semimonthly in December, by M&T Publishing, Inc., 501 Galveston Dr., Redwood City, CA 94063; 415-366-3600. Second-class postage paid at Redwood City and at additional entry points.

ARTICLE SUBMISSIONS: Send manuscripts and disk (with article and listings) to the editorial assistant 415-366-3600.

***DDJ* ON COMPUSERVE:** Type GO DDJ.

***DDJ* LISTING SERVICE:** 603-882-1599. Supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Type *listings* (use lowercase) at the login prompt.

SUBSCRIPTION: $29.97 for 1 year; $56.97 for 2 years. Foreign orders must be prepaid, including the additional postage (air or surface) in U.S. funds drawn on a U.S. bank. Add $13 per year for surface mail; add $36 for airmail to Canada and Mexico; or $26 for airlift to all other countries.

POSTMASTER: Send address changes to *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. **ISSN 1044-789X**

CUSTOMER SERVICE: For subscription questions, call toll-free 800-456-1215. For subscription orders or change of address call toll-free 800-456-1215 (U.S. and Canada) or write *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. For book/software orders call 800-533-4372 (in California 800-356-2002).

FOREIGN NEWSSTAND DISTRIBUTOR: Worldwide Media Service Inc., 115 E. 23rd St., New York, New York 10010; 212-420-0588 FAX 212-420-1265.

Entire contents copyright ©1990 by M&T Publishing, Inc., unless otherwise noted on specific articles. All rights reserved.

The Audit Bureau

can be recalled successfully. The pattern association is then permanently stored in this matrix.

Decoding, that is presenting one-half of a pattern and recalling the other half of the pair, is a bit more complicated. Because we now have several matrices storing pattern associations, we don't know which one is the correct one to look in to recall the pattern pair. To choose which pattern pair to recall from each matrix, we use the following criterion.

We determine all the returned pattern pairs (X_i,Y_i) that

If we are trying to encode

$A_1 = (101010)\ B_1 = (1100)$
$A_2 = (111000)\ B_2 = (1010)$

we first convert to {-1,+1}.

$X_1 = (1\ -1\ 1\ -1\ 1\ -1)$ $\quad Y_1 = (1\ 1\ -1\ -1)$
$X_2 = (1\ 1\ 1\ -1\ -1\ -1)$ $\quad Y_2 = (1\ -1\ 1\ -1)$

$$X_1^T Y_1 = \begin{matrix} 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ -1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 \end{matrix}$$

$$X_1^T Y_2 = \begin{matrix} 1 & -1 & 1 & -1 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \\ -1 & 1 & -1 & 1 \\ -1 & 1 & -1 & 1 \end{matrix}$$

$$M = \begin{matrix} 2 & 0 & 0 & -2 \\ 0 & -2 & 2 & 0 \\ 2 & 0 & 0 & -2 \\ -2 & 2 & 0 & 2 \\ 0 & 2 & -2 & 0 \\ -2 & 0 & 0 & 2 \end{matrix}$$

Example 1: *The encoding process*

Each neuron b_j in field Fb (Fa and Fb will be used to refer to the two pattern fields A and B) receives a gated input of all the neurons in Fa with a nonlinear threshold function applied. In our bipolar discrete example a typical function might be:

f(x,y) =1 if x > 0
y if x = 0
0 if x < 0

We now have a pattern B_1. However, we aren't done yet. The output from pattern B is then fed back through the transpose of matrix M to produce pattern A_1. That is, each neuron A_i in A receives gated input from each neuron B_j in B and applies the same threshold function to it.

A_1 is then sent back through the matrix again to produce B_2, and on this goes.

$A \rightarrow F(AM) \rightarrow B_1$
$A_1 \leftarrow F(B_1M^T) \leftarrow B_1$
$A_1 \rightarrow F(A_1M) \rightarrow B_2$
.
.
$A_i \rightarrow F(A_i) \rightarrow B_i$
$A_i \rightarrow F(B_i M) \rightarrow B_i$

Example 2: *The decoding process*

(continued from page 18)
have the same energy as the pair (A,Y_i) (where A is the presented pattern). Of these patterns we choose that pattern pair whose energy is closest to the matrix's orthogonal BAM energy. (Orthogonal BAM energy is the energy a matrix would have if all its stored patterns were orthogonal, which turns out to be equal to the negative of the product of the pattern lengths, $E^* = -np$. Energy of a pattern pair can be calculated the same way as in our previous discussions, $E = -XMY^T$, where X and Y are the two patterns.)

There are some problems with the Bam System. In order to keep checking that the patterns were stored reliably in each matrix (without corrupting the other patterns already in the matrix) the patterns need to be stored separately. Also, the need to compute the "best" recall from each of the BAM matrices could be computationally prohibitive. Parallel hardware (which, presumably, a BAM would be running on anyway) could possibly ease this burden.

The Implementation

C++ provides an excellent tool for implementing neural nets in general and BAMs in particular. Most of the constructs in this discussion of BAMs were vectors and matrices. This is a classic application of object-oriented programming. Classes for vectors and matrices should go a long way toward making the implementation easier. Listing One (page 84) is BAM.HPP, the BAM header file that contains the class definitions. Listing Two (page 84) is BAM.CPP, the BAM program file that contains the BAM implementation.

The vector class is implemented in classic fashion (almost identical to Stroustrup's). Methods are provided for assignment, multiplication by scalar constant, and dot product. This is all that is really necessary, but a few more methods are provided for completeness. Streams input and output are provided to read the patterns in and display patterns to the user. The streams functions do the necessary (0,1) to (-1, +1) conversion discussed earlier.

The matrix class is implemented as an array of pointers (int **), with indicators of the number of rows and columns. It could conceivably have been implemented as an array of vector objects. I chose representation for efficiency. There are several constructors provided. The first simply initializes the matrix from specified dimensions. These dimensions default to the particular application's two pattern lengths (specified by the ROWS and COLS constants). Other constructors are provided to form a matrix from a pair of vectors by multiplying one vector by the transpose of another ($M=AB^T$). Standard matrix arithmetic functions are included. Methods are also provided to form a vector from a row or column "slice" of the matrix. Streams output is provided for debugging diagnostics.

Another fundamental construct is the pattern pair. This is, after all, what the BAM lets us do — retrieve pattern associations. Pattern pairs are represented by the "vecpair" (vector pair) class. An "encode" operation will encode a vecpair. A "recall" operation will return a vecpair, when supplied with a pattern (or "vec").

Once we have these vector, matrix, and vector pair classes, implementing the BAM is fairly simple. The BAM is essentially just a matrix. We use the C++ inheritance mechanism to inherit the matrix and all its functions. We made the matrix's data structures "protected" instead of "private" so the derived BAM matrix class could use the matrix's data structures. We now just add a vecpair pointer for the pattern pair list and the BAM matrix functions.

(continued on page 24)

(continued from page 20)

These consist mainly of the "encode" and "recall" functions central to the BAM. Encode simply takes the "vecpair" corresponding to the association and adds it (with matrix add) to the current BAM. Recall "feeds" the presented pattern through the matrix (with dot products and by applying a threshold function as discussed earlier) to return another vector. We keep feeding the vectors back and forth until they stabilize to a consistent pattern association. There are also some auxiliary functions for checking the integrity of the BAM, returning its energy for a particular association (as discussed earlier), and for "uncoding" or removing an association from the BAM.

The Bam System class consists of an array of pointers to

BAMs are a neural-network-based attempt at content-addressable memories

BAM matrices. Each time a BAM matrix is saturated, a new matrix is created, and the new pattern association is stored in it. The major functions are again "encode" and "recall." Encode attempts to store the pattern association in each of the BAM matrices until it succeeds. It will create a new BAM matrix if it runs out of matrices. Recall performs a BAM matrix recall operation on each of the BAM matrices. The returned association that is closest to the presented pattern and has the lowest energy relative to its matrix (as discussed earlier) is then returned as the "correct" pattern association. Another function is provided to "train" the Bam System from a specified file of pattern associations. The patterns happen to be represented as 01 strings, but this could be easily changed to whatever representation (for example, floating-point numbers, character strings) suits the specific application.

Thanks to the wonders of C++, the code is very readable. Most of the algorithms can be implemented in the same vocabulary as the theory. Take a look at it to examine the mechanics in detail. It should even be clearer (and certainly more specific) than the discussion above.

The Test Program

I've included a test program (TESTBAM.CPP, Listing Three, page 88) that demonstrates an actual running Bam System. A Bam System is created and told to "train" itself from the file TEST.FIL (see Listing Four, page 88). This file contains a set of simple "pattern pairs," represented as (0,1) strings delimited by commas — one pattern pair to a line. Once the Bam System is trained, you can enter any pattern you want (using the 01 format mentioned) and the correct pattern association will be recalled. If the pattern is slightly wrong, the correct pattern association will still most likely be recalled. The make file, TESTBAM.MK (Listing Five, page 88), shows how to construct this test program.

What Can you Do With It?

Uses of the Bam System are constrained only by your imagination. Obvious uses include optical character recognition (the pixel patterns scanned in would be associated with the actual letters), voice recognition (the acoustic pattern would be associated with the actual word), or a super spell checker (word patterns associated with phoneme-string patterns). You can use a Bam System in just about any

(continued from page 24)
application where you have a large number of "associations" that you would like to be able to recall close to instantaneously, and where some tolerance for error would be useful.

A successful application of BAM for radar signature classification was presented at the January 1990 International Joint Conference on Neural Networks (IJCNN). However, it was not a Bam System, and the implementors had to resort to various other tricks to get around capacity limitations. Several other associative memory applications appeared; but none of them were associative memory systems. They all would probably run into the capacity roadblock eventually for large data sets. Associative memories and BAMs have begun to appear implemented in VLSI, but again the capacity will prove to be a limitation for practical work. Bam Systems should have a radical effect on the usefulness of these chips.

Conclusion

Bidirectional associative memories appear to provide the content-addressable memory long sought after by computer scientists. They provide instant recall of pattern association, tolerance for error and fuzziness in the provided pattern, and global stability. However, by themselves they face some limitations. Simple BAM matrices cannot encode more pattern pairs than the smaller of their two dimensions. Some applications have inherently smaller pattern length, and, for them, matrix capacity will prove to be a severe limitation. However, the Bam System appears to overcome this problem, making associative memory a reality.

Notes

Grossberg, S. *The Adaptive Brain*, I & II. Boston, Mass.: Reidel Press, 1982.

Kohonen, T. *Self-Organization and Associative Memory*. Berlin, W. Germany: Springer-Verlag, 1977.

Kosko, B. "Bidirectional Associative Memories," IEEE Trans. Systems, Man, Cybernetics, Vol. SMC-L8, 49-60, Jan./Feb. 1988.

Rumelhart, D.E., McClelland, J.L., eds., *Parallel Distributed Processing*, I & II. Cambridge, Mass.: MIT Press, 1986.

Simpson, P.K. "Bidirectional Associative Memory Systems," *Heuristics*, 1989.

Simpson, P.K., "Associative Memory Systems," *Proceedings of the International Joint Conference on Neural Networks*, January 1990.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 84.)

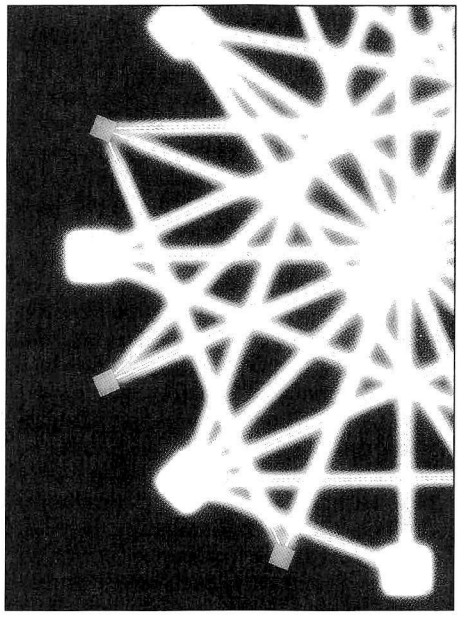
Vote for your favorite feature/article.
Circle Reader Service **No. 1.**

A Neural Network Instantiation Environment

Dynamically creating neural nets lets you concentrate on network response characteristics

Andrew J. Czuchry, Jr.

The automatic generation of tailored neural network architectures greatly simplifies the tedious task of putting together neural networks. Typically, an architecture is assembled by manually writing and modifying a collection of software routines; automation speeds this standard process of assembling networks. However, task simplification through the automatic generation of network architectures often implies limited flexibility when applied to real-world problems. In order to develop useful network architectures in an efficient manner, the provision of both task simplification and complete flexibility is an inherent design principle in the research environment that instantiates (dynamically creates) neural networks. Instantiation is the flexible process of automatically piecing together architectures based upon modifiable structures that represent the parameters of the assembled neural networks.



The incorporation of network instantiation into an entire research environment for neural networks results in a system that provides both task simplification and complete flexibility.

Task simplification can be achieved by using a variety of knowledge-representation techniques. Complete flexibility can be maintained by strictly applying standard software-modularization techniques. The merging of these two types of techniques — knowledge representation and software modularization — provides the foundation for the instantiation process that forms the basis of a powerful neural network research environment.

In this article, I discuss the need for such an environment and describe a working version. In so doing, I describe the knowledge-representation techniques used, and the essential integration of knowledge representation and software modularization. (The model was developed on a Symbolics Lisp machine, chosen for its flexibility and power in symbolic manipulation and for its exploratory programming environment. The implementation language is Lisp.) I also present experimental results of using the environment for a test-case network, and finally, I discuss future efforts and the evolution of the environment.

System Overview

The task of generating usable neural network architectures for real-world problems is quite challenging. Basic standard networks are merely skeletons for useful systems. For example, Fukushima's neocognitron[1] is really a class of neural networks. Most often only specific instances (class elements)

Andy earned the A.B. degree in computer science from Dartmouth College and the M.S. degree in information and computer science from the Georgia Institute of Technology. He is presently pursuing a Ph.D degree in information and computer science at the Georgia Institute of Technology. His research is supported by the Artificial Intelligence Branch of the Georgia Tech Research Institute. Andy has published several articles on topics concerning "intelligent" computer systems. He can be reached at the Georgia Institute of Technology, A. I. Branch, Georgia Tech Research Institute, 243 Baker Bldg., Atlanta, GA 30332.

(continued from page 28)
are described in the literature — the skeleton for a neocognitron is a multilayer, hierarchical neural network for visual pattern recognition. It consists of a series of layers of subnetworks that are organized according to specific guidelines. The system's exact parameters (for example, the number of layers, the number of subnetworks per layer, and the size of each subnetwork) are often tailored to the problem being addressed. This tailoring is the meat on the skeleton and is determined by the application's processing requirements.

Such tailoring is evident in the differences between the architectures of the neocognitron described by Fukushima and Miyake[2] and by Fukushima.[10] Fukushima and Miyake[2] describe a seven-layer system for type-written or stylized (that is, written to meet certain specifications of consistency) numeral recognition. Each layer of the network has 24 subnetworks, except for the input layer, which is a single subnetwork layer. In contrast, the architecture of the neocognitron for hand-written numeral recognition, as described in Fukushima,[10] is a nine-layer network with 1, 12, 8, 38, 19, 35, 23, 11, and 10 subnetworks per respective layer.

In addition to these architectural differences, the setting of various internal parameters may also vary according to the application. More noise tolerance is provided by decreasing the inhibitory ("negative") weights and shrinking the number of connections per node in a subnetwork. Finer degrees of class separation are provided by increasing inhibition and increasing the number of connections between the nodes in each subnetwork. A variety of other internal parameters can be altered as well.

Given the goal of efficiently establishing useful architectures and parameter settings for real-world applications, automatic generation of neural networks based upon a flexible representation of the desired characteristics is vital. This goal has been realized through the development of the research environment described in this article. The environment dynamically creates neural networks based upon the information encoded in underlying knowledge-representation structures. The research environment automatically builds these structures based upon parametric specification of the desired characteristics of the network architecture.

For example, passing the network creation routines the network type of *neocognitron*, the layer number *9*, and the subnetwork size list of *(1, 12, 8, 38, 19, 35, 23, 11, 10)* would produce an architecture similar to the one described by Fukushima.[10] An exact match to Fukushima's architecture could be obtained through additional parameter specifications. The key point is that the research environment comprises a combination of multipurpose routines that are pieced together appropriately through the use of flexible knowledge representation structures. The environment's flexibility is maintained through the strict application of software-modularization techniques. For example, software modularization ensures that weight calculation routines can be adjusted independently of the connection calculation routines. These ideas are clarified in the following sections.

Knowledge Representation

There are two fundamental ideas behind the use of knowledge representation. The first is that simple parametric changes can significantly alter the network architecture's final structure. For example, changing the size (number of nodes) in each subnetwork can greatly affect the specific connections between the nodes. This is of primary importance in networks such as the neocognitron[1] for two reasons:

1. The connections are between subnetworks rather than within subnetworks.
2. The nodes are not completely connected (that is, every node is connected to only a subset of the nodes in other subnetworks). This means that the connection architecture is heavily influenced by the size and number of subnetworks.

The second fundamental idea is that many routines for the creation of neural networks and subsequent network processing are common to entirely different architectures. As a result, these routines can be reused and, to some degree, tailored automatically by combining and adapting the modules. The realization of a research environment that automatically generates flexible neural network architectures has, thus, been based upon a knowledge representation in which every structure "carries around with it" all the local information for piecing itself into the network puzzle and for subsequently computing/processing data once the architecture is assembled.

Three main knowledge structures — *NET*s, *LAYER*s, and *PLANE*s — collectively compose the knowledge representation. These structures are presented in Listing One, page 93. The *NET* structure consists of a list of one or more layers and a variety of local parameters specific to the global processing of the particular type of architecture (for example, the vigilance parameter in ART.[3,5]) *LAYER* encodes a list of subnet-

(continued from page 30)
works and the connections between layers (e.g., a list of the connections from each subnetwork in the present layer to the subnetworks of the preceding layer). Local parameters, such as inhibition constants or gain constants, are also stored within the layer. The type of the layer (for example, S or C for the neocognitron[1] and F_1 or F_2 in ART[3,5]) is also recorded. A pointer to the previous layer is provided so that the routines can "get around" in the network. *PLANE*, a subnetwork, is used to store the connections within the subnetwork. The weights for both inter- and intraplane connections are also recorded in the *PLANE* structure. A *size* parameter for the plane is used for instantiation and is locally encoded. A pointer to the layer of which this plane is a part is stored as well. The actual nodes (cells) or processing elements are stored as an array, which is used to record output activation values.

In the future, this array will be extended so that each node is itself a knowledge structure. In this way, the activation functions, output functions, and local node parameters can be maintained locally. Such an extended representation will further increase the environment's flexibility by providing for the adjustment of input and output activation functions within the overall architecture and, thereby, will extend the standard of common activation functions for each cell in the subnetwork. This standard for specific activation functions for the entire subnetwork is not a severe restriction. However, the extended representation will support novel research endeavors and, thus, could prove to be extremely valuable.

The information contained in these knowledge structures is stored at the time of network instantiation and is utilized as the computational map by the processing routines. The structures thus dynamically control the routines to be called, the data to be passed, and the amount of processing to be performed. Each structure has been designed to carry locally all the information necessary to direct processing through the contents of the structure itself rather than through a priori routines. This advantage increases processing flexibility. In adapting the flow of processing, no routines need to be altered; only the information in the structures is modified.

Modularity

Great care has been taken to ensure software modularity. The significance of this is apparent upon analysis of the power obtained by the integration of knowledge representation and software modularization. Before discussing the integration, however, I will briefly describe the modularity.

Software modularity has been preserved in all three of the main phases of neural network applications: instantiation, processing, and training. The network instantiation routines, highlighted in Listing Two, page 93, are the pieces that are meshed to dynamically create network architectures. Instantiation is obviously the first step in the use of neural networks for any application. The instantiation process proceeds from generic net-level creation routines to specific layer-level creation routines that are tailored to the specific type of network. Finally, it proceeds to generic plane-level routines that perform connection and connection weight calculations in addition to creation of the plane itself. Upon completion of the instantiation process, the network is ready for training.

Listing Three, page 96, indicates the training routines. These routines are used to perform the processing all the way from the network level down to the level of the individual cell. Listing Three is abbreviated, however, and depicts the routines only down to the initial processing at the plane level. Subsequent processing occurs at the plane, connection, and cell levels. After training, the network can be used for identification tasks. Identification functions at the network level are presented in Listing Four, page 98. Further processing occurs at the layer and plane levels but is not included here.

Integration

The knowledge representation structures function as generic placeholders in which data about instantiated neural networks is recorded. As mentioned previously, the instantiation process dynamically produces an entire network based on the parametric specification of the desired characteristics. Instantiation begins by calling *CREATE-NET* (see Listing Two) and passing it the appropriate parameters for the desired network.

Two examples of the parametric settings and function calls for different versions of a neocognitron are depicted in Listing Five, page 98. Evaluating **neocognitron-net**, (that is, *(eval *neocognitron-net*)*) returns a *NET* structure (see Listing One) that contains an instantiated network meeting the characteristics specified in the parameters recorded in the **neocognitron-net** variable. More specifically, a seven-layer network would be created. The input layer would contain one subnetwork (plane), and each of the other layers would contain 24 planes. The plane in the input layer would contain a 16×16 array of nodes (cells). Each of the planes in the next layer would contain a 16×16 array of cells. Subsequent layers would be composed of planes with 10×10, 8×8, 6×6, 2×2, and 1×1 arrays of cells, respectively.

Each cell in a plane would have a "square" projection pattern; cells are connected to other cells that occupy a corresponding square area in another plane. Connections from the first layer to the input layer would cover a 5×5 array. Connections from the second layer to the first would also cover a 5×5 array, and similarly for all the connections up to and including the connections from the sixth layer to the fifth layer. Connections between the last (seventh) and the sixth layer, however, would cover a 2×2 array. Each cell, x_i, would thus become an input to multiple other cells, x_j, and each x_j would receive inputs from many different x_k cells.

Each of these connections has associated with it a connection weight. Within the knowledge representation, the connection weights are stored separately from the connections themselves so as to provide for adaptation of the weights independently of the connection structure. In addition to these elements common to all neural networks (that is, one or more layers, one or more subnetworks per layer, individual cells, connections, and connection weights), a variety of other parameters significant for the neocognitron would be set as indicated in the **neocognitron-net** variable in Listing Five.

An additional comment about the assignment of connections is important. The exact connection patterns are calculated based upon the size of the sending and receiving planes and the size of the projection area. The instantiation routines are structured such that the appropriate connections are computed based upon the parameters passed to the respective routines. For example, if each element in a 5×5 network is to project into a 3×4 network such that each element has a 2×2 projection and all of the 3×4 network elements are covered, then the system would compute that the element in position (0,0) of the 5×5 network would be connected to the elements in positions (0,0), (0,1), (1,0), and (1,1) of the 3×4 network (see Figure 1). The element in position (4,4) of the 5×5 network would be connected to the elements in positions (1,2), (1,3), (2,2), and (2,3) of the 3×4 network. A variety of standard connection architectures have been presented in

(continued from page 32)
the neural network literature — for example ART,[3,5,6,7] back propagation,[8] Hopfield networks,[9] and the neocognitron.[1] Because the connection calculation routines are parameterized and actually calculate the connection patterns, arbitrary algorithmically expressed connection patterns can be realized.

Experimental Results

To investigate the viability of the research environment presented in this article, a standard neural network architecture was chosen to test the environment's instantiation, training, and identification capabilities. The network chosen was the neocognitron[1] because of its large size and the complexity of the connection architecture. More specifically, the architecture presented by Fukushima and Miyake[2] was reproduced and is characterized by **neocognitron-net** in Listing Five. For this version of a neocognitron, there are a total of more than 2.3M connections, each with its own weighting factor, between the 10K cells and 145 planes in the network. Additionally, several parameters interact and affect the behavior of the network. For example, each of the layers (excluding the input layer) has an intensity-of-inhibition parameter to control the amount of noise tolerated in matching a pattern; this parameter interacts with both the excitatory and inhibitory weights as an output is computed for a particular network cell.

Instantiation of the network, described by the **neocognitron-net** variable in Listing Five, and subsequent training and identification testing yielded significant results:

1. Different patterns produce different excitation patterns within the network (see Figures 2 and 3).
2. Training the network alters its excitation patterns (see Figures 3 and 4).
3. After training, only a single cell fires at the recognition layer in response to different stimulus patterns (Figure 4).
4. Appropriate clusterings are achieved for multiple versions of various numeric characters (see Figures 4, 5, and 6).

The input pattern is depicted at the base of each figure in Figures 2 through 5. Each layer is represented by the double row of squares (planes), which are respectively labeled S_1, C_1, S_2, C_2, S_3, and C_3 on the right-hand edge of the figures. The colored areas within each square represent the output activities of the corresponding nodes (cells). The color scale is shown on the left-hand edge of the figures and indicates that

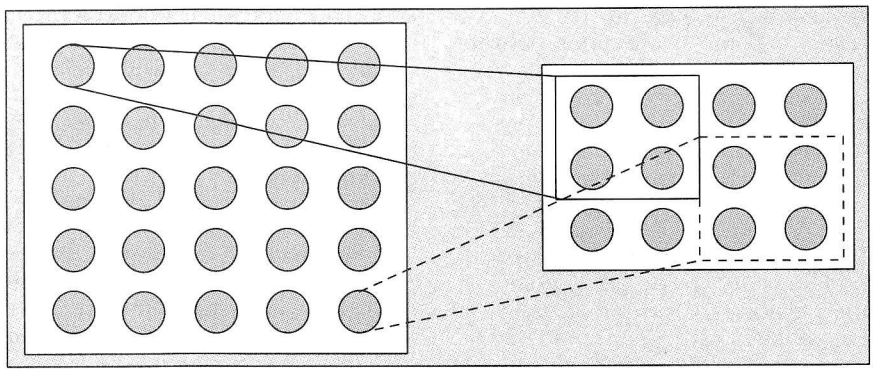


Figure 1: *Connections calculated for 2× 2 projections from a 5× 5 network into a 3× 4 network*

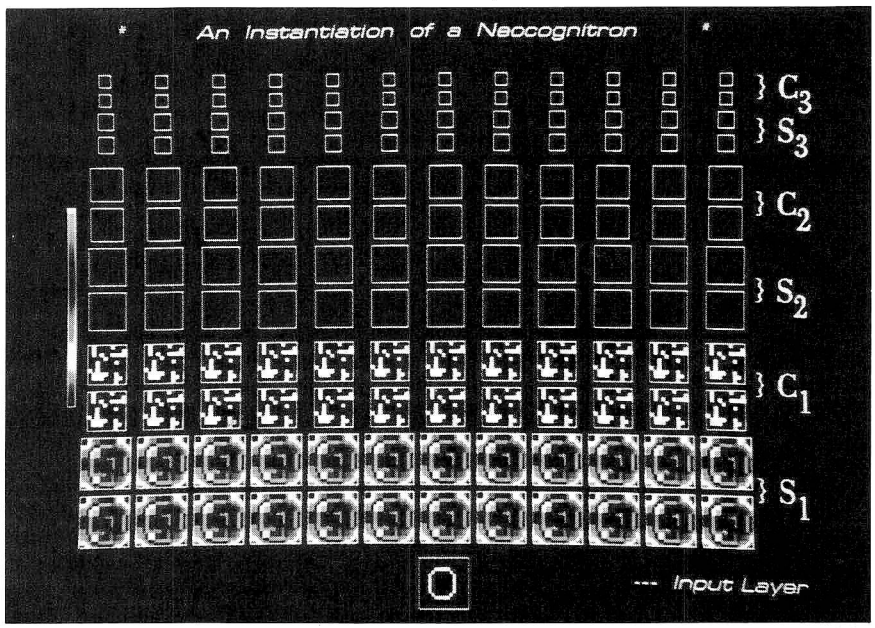


Figure 2: *Response characteristics of an instantiated neocognitron to the input pattern 0 before training*

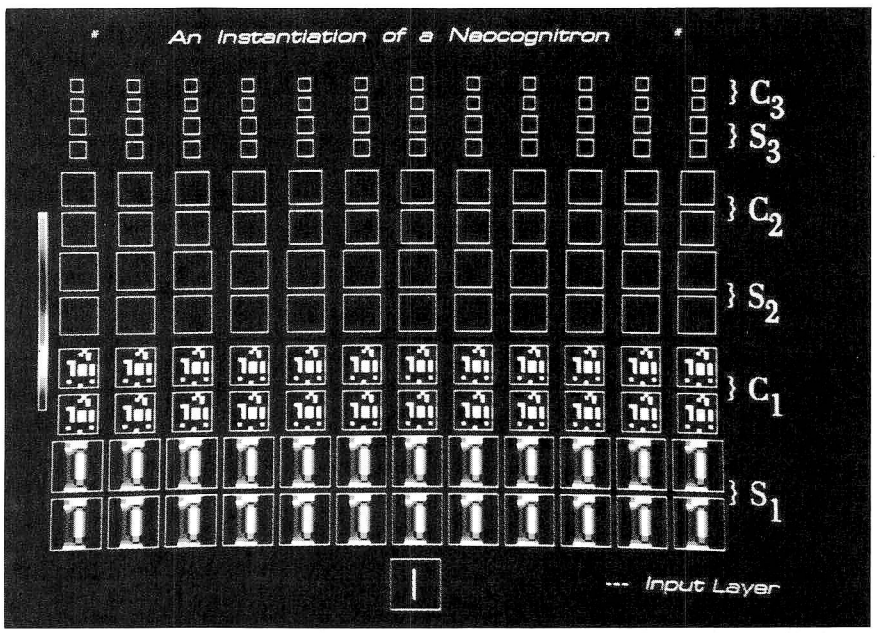


Figure 3: *Response characteristics of an instantiated neocognitron to the input pattern 1 before training*

activity ranges from a low level of black to blue, to green, to red, to yellow, to a high level of white. These pictures are produced within the research environment as a useful utility for qualitatively observing the results of the particular instantiated architecture. As depicted herein, the results correlated well with those presented by Fukushima and Miyake.[2]

Future Efforts

The research environment and corresponding instantiation of neural networks have many possible applications. From a general perspective, such applications encompass both new-model development and the analysis of standard network models. More specifically, one of the motivating ideas behind this research has been that of using digitized images as training patterns. The hypothesis is that network models such as the neocognitron should theoretically be able to extract "useful" information from such images. For example, through training an instantiated network using a variety of images that contain a tree, the network should extract the common pattern of the tree and, thus, be able to indicate the presence of a tree in subsequent test images.

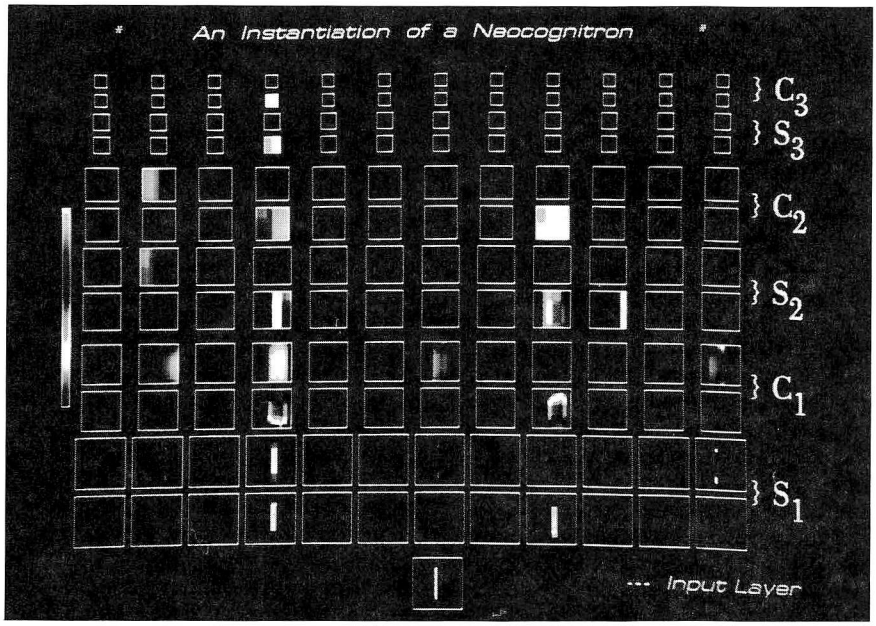


Figure 4: *Response characteristics of an instantiated neocognitron to the input pattern 1 after training*

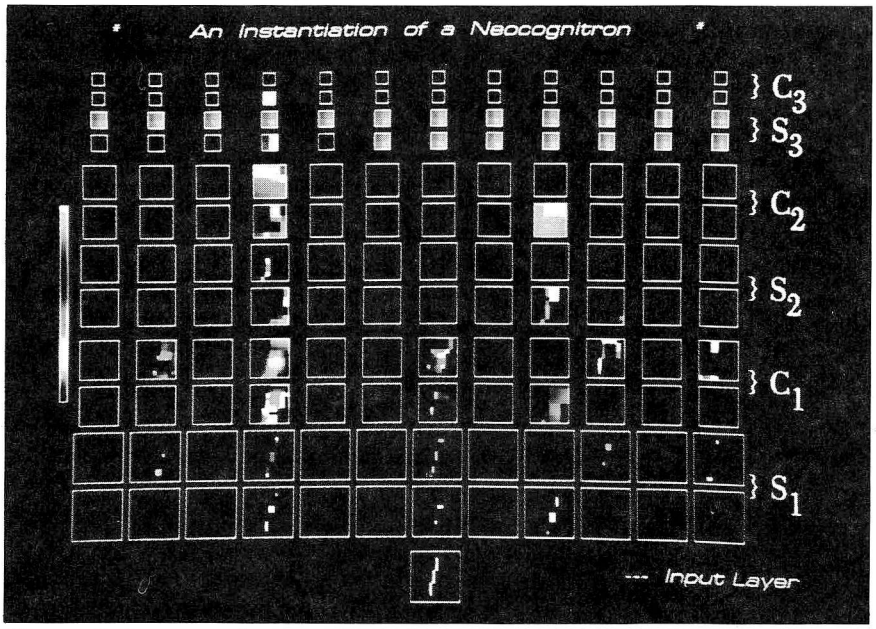


Figure 5: *Response characteristics of an instantiated neocognitron to the input pattern of a slanted 1 after training*

A possible future research effort would investigate the size of various networks required to actually perform such recognition and to characterize any additional requirements (for example, use of Grossberg's Boundary Contour/Feature Contour System [S. Grossberg and E. Mingolla, "Neural dynamics of perceptual grouping: Textures, boundaries, and emergent segmentations," Perception & Psychophysics, 38 (1985), 141 – 171.] as a preprocessor to simplify processing within an appropriate version of a neocognitron). Additionally, recent extensions to the neocognitron's architecture (that is, feedback between layers[3]) could be incorporated into the currently instantiated neocognitrons and could possibly provide for segmentation of trees within the test images after training has occurred. A significant amount of work would be required to obtain such results, but a real possibility of attaining them does exist.

On the front of run-time analysis of instantiated networks, the speed and versatility of processing related to the implementation of the environment should be considered. As mentioned, the present implementation, which was developed for in-house use, was developed on a Symbolics Lisp machine. The lisp machine was chosen for its flexibility and power in symbolic manipulation and for its exploratory programming environment. The implementation language is Lisp. In order to enhance processing speed, there is a plan to port the environment to a Sun 4/280. Although the environment is currently organized to dynamically create Fukushima's neocognitron,[2] its versatility will be tested by instantiating additional neural network models.

Conclusion

The main virtue of the environment described here is that it frees the user/programmer/researcher from the need to write programs that assemble neural networks; the environment automatically generates flexible neural network architectures based upon parametric specifications. Flexibility is achieved through the integration of knowledge-representation and standard software-modularization techniques. Together, these two types of techniques form the powerful basis of a research environment for neural networks.

The mechanisms through which the power is harnessed and utilized are the heart of this article. The key idea is that a knowledge representation has been developed so that each element of a neural network carries around with it all the information necessary for local processing (for example, what processing to perform and where to get the inputs). Because this information storage is consistent from the node level to the network level, the entire network is executed without the need for global routines to encode its structure. Generic routines become specialized processors as they are adapted by the contents of the knowledge structures.

The viability of such an environment has been demonstrated through the instantiation and testing of a standard network architecture, the neocognitron. The results correlate accurately with those of an analogous network described by Fukushima and Miyake. The present work suggests many significant applications, some of which are currently under investigation.

Knowledge representation and software modularization are key tools ideally suited for the empirical analysis of neural networks. Wrapping an environment around these fundamental tools facilitates concentration on network-response characteristics rather than on monotonous debugging of specialized routines that encode network architectures.

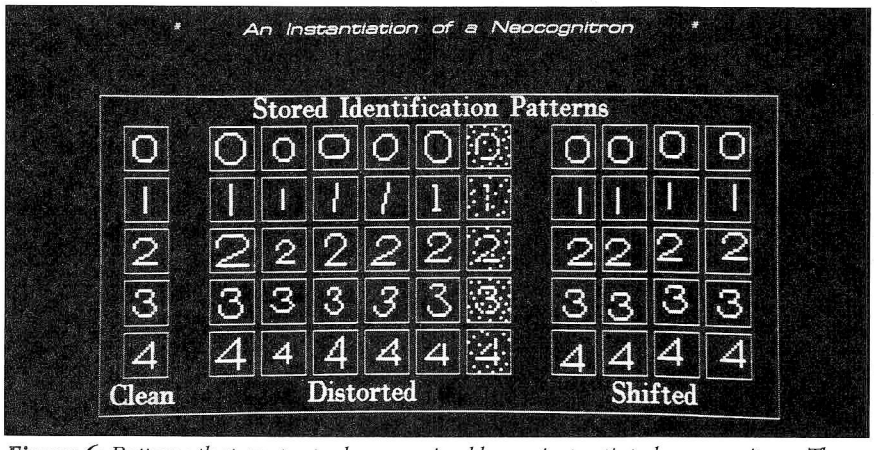


Figure 6: *Patterns that are properly recognized by an instantiated neocognitron. The neocognitron was first trained on the patterns in the leftmost column*

Acknowledgments

This work has been performed under the guidance and with the support of John Gilmore, head of the Artificial Intelligence Branch at the Georgia Tech Research Institute. Additional contributions to the initial design and development of this environment have been made by Harold Forbes and Steven Strader. I am indebted to Diane Czuchry for her assistance in the preparation of this manuscript.

Notes

1. K. Fukushima, "Neocognitron: A self-organizing neural network for a mechanism of pattern recognition unaffecte by shift in position," *Biological Cyb netics* 36 (1980) : 193 – 202.
2. K. Fukushima, and S. Miyake, " cognitron: A new algorithm for recognition tolerant of deformat shifts in position," *Pattern Re* 15 (1982) : 455 – 469.
3. G. Carpenter, and S. Grossberg, "A massively parallel architecture for a self-organizing neural pattern recognition machine," *Computer Vision Graphics Image Processing* 37(1) (1987) : 54 – 115.
4. K. Fukushima, "A neural network for visual pattern recognition," *IEEE Computer* 21(3) (March 1988) : 65 – 75.
5. G. Carpenter, and S. Grossberg, "ART2: Self-organization of stable category recognition codes for analog input patterns," *Applied Optics*, 26 (1987) : 4919 – 4930.
6. S. Grossberg, "Adaptive pattern classification and universal recoding: I. Parallel development and coding of neural feature detectors," *Biological Cybernetics* 23 (1976) : 121 – 134.
7. S. Grossberg, "Adaptive pattern classification and universal recoding: II. Feedback, expectation, olfaction, illusions, *Biological Cybernetics* 23 (1976) : 187 – 202.
8. D. Rumelhart, G. Hinton, and R. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing* 318-362. (Cambridge, Mass.: MIT Press, 1986.)
9. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Poc. Nat. Academy Sci, USA* 79 (1982) : 2554 – 2558.
10. K. Fukushima, "Neocognitron: A hierarchical neural network capable of visual pattern recognition," *Neural Networks* 1 : (1988) 119 – 130.

DDJ

(Listings begin on page 93.)

Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

Untangling Neural Nets

When is one model better than another?

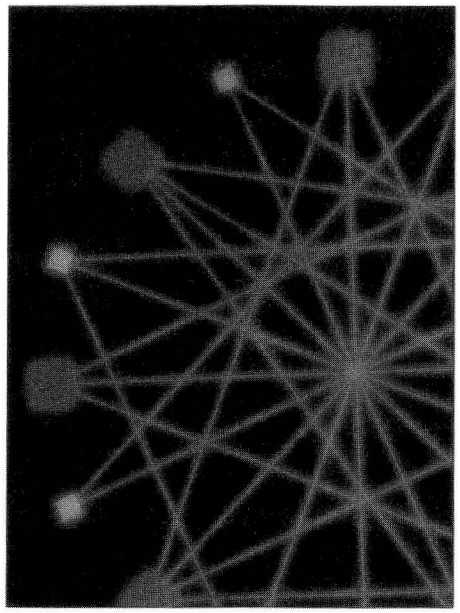
Jeannette "Jet" Lawrence

Neural networks, which are formed by simulated neurons connected together much the same way the brain's neurons are, are able to associate and generalize without rules. They have been used to classify undersea sonar returns, speech, and handwriting, predict financial trends, evaluate personnel data, control robot arms, model cognitive phenomena, and much more.

The kinds of problems best solved by neural networks are also those that people do well: Association, evaluation, and pattern recognition. Neural networks also handle problems that are difficult to compute and do not require perfect answers — just quick, good answers. This is especially true in real-time robotics or industrial controller applications.

Other appropriate applications are predicting behavior and analyzing large amounts of data, such as in stock market forecasting and consumer loan analysis. New applications under development include simple vision systems, weather forecasting, assistance in medical diagnosis, and estimation of the worth of insurance claims.

Jeannette (Jet) Lawrence is technical publications manager at California Scientific Software, and the author of their 1989 publication Introduction to Neural Networks. *She can be contacted at 160 E. Montecito #E, Sierra Madre, CA 91024.*



A neural network is not always the best solution for certain problems. They are poor at precise calculations and serial processing, nor are they able to predict or recognize anything that does not inherently contain some sort of pattern. This is why, for example, a neural net cannot predict the lottery, because a lottery is by definition a random process.

It is unlikely that a neural network could be built that has the capacity to think as well as a person does for two reasons: Neural networks are terrible at deduction (logical thinking), and the human brain is too massively complex to simulate completely. A human brain contains about 100 billion neurons, each of which connects to about 10,000 other neurons.

A brief look at the general structure and operation of neural networks will help explain the limits of neural networks abilities. There are many types of neural networks, but all have three things in common: Distributed processing elements (neurons), the connections between them (network topology), and the learning rule. These three aspects together constitute the neural-network paradigm.

The Formal Model of a Neuron

Artificial neurons are also known as processing elements, neurodes, units, or cells. Figure 1 shows the canonical model of a neuron. Each neuron receives the output signals from many other neurons. The point where two neurons communicate is called a "connection." This neural connection is analogous to a biological synapse in the mammalian brain. A neuron calculates its output by finding the weighted sum of its inputs. The strength of a particular connection, called its weight, is noted w_{ij}, where i is the receiving neuron and j is the sending neuron.

At any point in time (t), the activation function, adds up the weighted

(continued from page 38)
inputs to produce an activation value $a_i(t)$. In most models, input signals can either be excitatory or inhibitory, that is, they either tend to make the neuron fire or tend to suppress its firing. This value is passed through an output (or transfer) function f_i, which produces the actual output for that neuron for that time, $o_i(t)$.

After summation, the net input of the neuron is combined with the previous state of the neuron to produce a new activation value. In the simplest models, the activation function is the weighted sum of the neuron's inputs; the previous state is not taken into account. In more complicated models, the activation function also uses the previous output of the neuron, so that the neuron can self-excite. These activation functions slowly decay over time; an excited state slowly returns to an inactive level. Sometimes the activation function is stochastic, that is, it includes a random noise factor.

The transfer function of a neuron defines how the activation value is output. The earliest models used a linear transfer function. However, certain problems are not entirely reducible by purely linear methods. The threshold transfer function is the simplest of the nonlinear models. This function is an all-or-nothing function; if the input is greater than some fixed amount (the threshold), the neuron will output a 1; if the value is below the threshold, the neuron will output a 0.

Sometimes the transfer function is a saturation type of function: More excitation above some maximum firing level has no further effect. A particularly useful transfer function is called the "sigmoid function," which has a high- and a low-saturation limit and a proportionality range in between. This function is 0 when the activation value is a large negative number. The sigmoid function is 1 when the activation value is a large positive number and makes a smooth transition in between.

The behavior of the network depends heavily on the way the neurons are connected. In most models, the individual neurons are grouped into layers so that the output from each neuron

The transfer function of a neuron defines how the activation value is output

in one layer is fully interconnected with the inputs of all the neurons in the next layer. A network may include inhibitory connections from one neuron to the rest of the neurons in the same layer called "lateral inhibition." Sometimes a network has such strong lateral inhibition that only one neuron in a layer, usually the output layer, can be activated at a time. This effect of minimizing the number of active neurons is known as "competition." In a feed-forward network, neurons in a given layer do not take inputs from subsequent layers or from layers prior to the immediately previous layer. Also, the neurons in a feed-forward network usually do not connect to each other. The back propagation network typically has three feed-forward layers: Input, hidden, and output. Feedback models additionally include connections from the outputs of one layer to the inputs of the same or a previous layer.

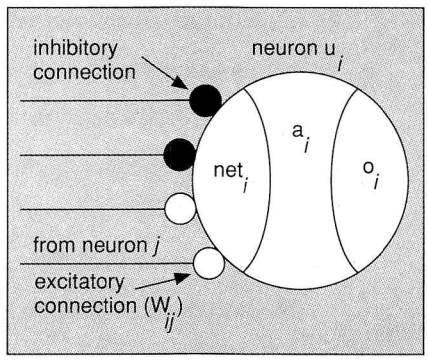


Figure 1: *The formal model of a neural-network processing element*

A neural network learns by adapting to changes in the input. This is accomplished through changes in the weights as the network gains experience. The learning rule is the very heart of a neural network; it determines how the weights are adjusted as the neural network gains experience. Of the numerous learning rules in use, the most well-known are Hebb's Rule and the Delta Rule. Nearly all other rules are variations of these two.

More than 30 years ago, Donald O. Hebb theorized that biological associative memory lies in the synaptic connections between nerve cells, and that the process of learning and memory storage involved changes in the strength with which nerve signals are transmitted across individual synapses. Hebb's Rule states that pairs of neurons that are active simultaneously become stronger by synaptic (weight) changes. The result is a reinforcement of those pathways in the brain. Hebb's Rule states $\Delta w_{ij} = \nu a_i o_j$ where ν is the learning rate that specifies a scaling factor for changes during training.

The Delta Rule, a supervised learning algorithm, additionally states that if there is a difference between the actual output pattern and the desired output pattern during training, then the weights are adjusted to reduce the difference. The Delta Rule states $\Delta w_{ij} = \nu(t_i - a_i)o_j$, where t_i is the training (desired output) pattern. The back-propagation rule is a generalization of the Delta Rule for a network with hidden neurons.

The best learning rule to use with linear neurons is the Delta Rule. This allows arbitrary associations to be learned, provided that the inputs are all linearly independent. Other learning rules (such as Hebb's) require that the inputs also be orthogonal.

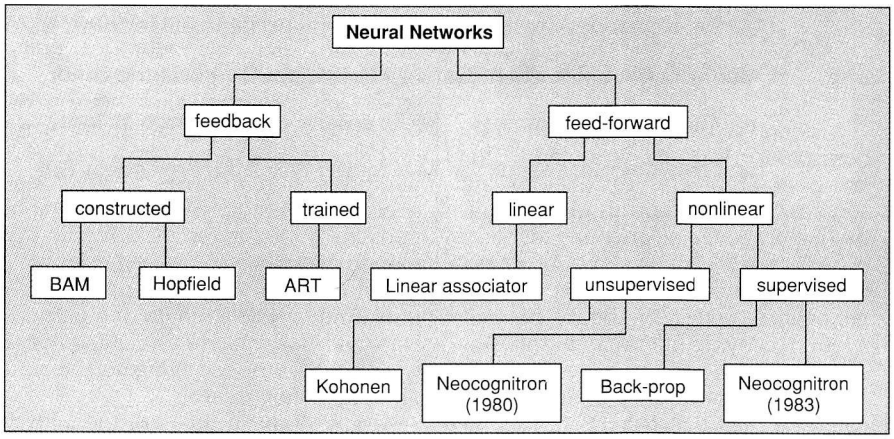


Figure 2: *A taxonomy of neural-network types*

The Two Major Topologies

Neural networks can be arbitrarily categorized by topology, neuron model, and training algorithm. (Figure 2 shows one method of classifying neural networks.) There are two main subdivisions of neural network models: Feed-forward and feedback topologies.

Feedback models can be constructed or trained. In a constructed model the

(continued from page 40)
weight matrix is created by taking the outer product of every input pattern vector with itself or with an associated input, and adding up all the outer products. After construction, a partial or inaccurate input pattern can be presented to the network, and after a time the network should converge so that one of the original input patterns is the result. Hopfield and BAM are two well-known constructed feedback models.

The Hopfield network is a self-organizing, associative memory. It is the canonical feedback network. It is composed of a single layer of neurons that act as both output and input. The neurons are symmetrically connected ($w_{ij} = w_{ji}$). (See Figure 3.) Hopfield networks are made of nonlinear neurons capable of assuming two output values: -1 (off) and +1 (on). The linear synaptic weights provide global communication of information. In spite of its apparent simplicity, a Hopfield network has considerable computational power.

The weight matrix is created by taking the outer product of each input pattern vector with itself, and adding up all the outer products. After construction, a pattern is given to the network. A process of reaction-stimulation-reaction between neurons occurs until the network settles down into a fixed pattern called a "stable state." Thus, the network result comes as a direct response to input.

The energy required by a device to reach a stable state can be plotted in three dimensions as a curved surface. In this representation, the stable states of the system (the energy minimums) appear as valleys. A neural network, which is used to find "good enough" solutions to optimization problems, may have many possible energy minimums or valleys. Depending upon the initial state of the network, any of the deepest valleys may end up as the answer. Inputing incomplete information to an associative memory network causes the network to follow paths to a nearby energy minimum where the complete information is stored.

Hopfield networks can recognize patterns by matching new inputs with the closest previously stored patterns. Hopfield networks are especially good for finding the best answer out of many possibilities. They are also good at recalling all of a stored piece of information when given partial data. Hopfield networks are often used in applications requiring some form of content addressable memory.

While the Hopfield model is able to associate on a large scale, it does not learn; the weights must be set in advance. A serious limitation of the Hopfield model is that the maximum number of memories M, which can be stored while still retaining perfect recall is [M less than or equal to N/(4 log N)] where N is the number of neurons. If more memories are stored, then the stable states begin to differ significantly from the stored information and eventually all will be forgotten. If an error rate of 5 percent is tolerable, then the capacity is about 14 percent of N. The hardware efficiency is also poor. A variation has been proposed, called the "Unary or Hamming" network, which uses inhibitory lateral connections in the internal neurons. It is claimed that this model has a capacity of M >> N with no errors in the final state.

A neural network learns by adapting to changes in the input

Bart Kosko brought the Hopfield network to its logical conclusion with the BAM. The BAM (bidirectional associative memory) is a generalization of the Hopfield network. Instead of creating the weight matrix with the dot product of a pattern with itself (auto-association), pairs of patterns are used (pair association). After construction of the weight matrix, either pattern can be applied as input to elicit as output the other pattern in the pair.

A trained feedback model is much more complicated because adjustment of the weights affects the signals as they move forward as well as back-

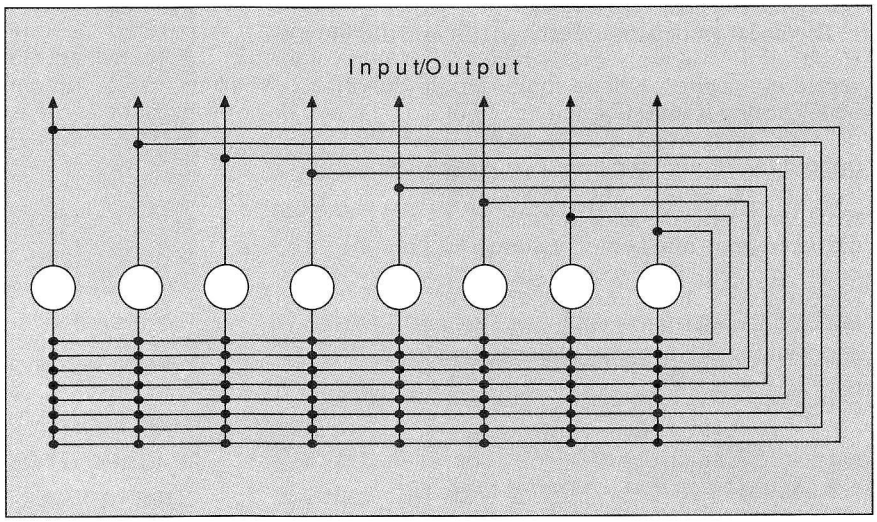


Figure 3: *The topology of a Hopfield neural network*

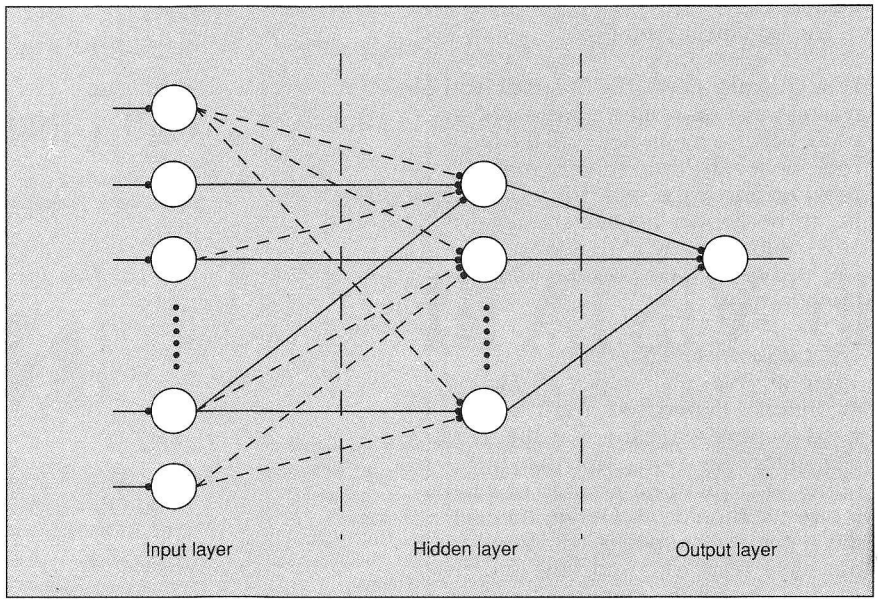


Figure 4: *The topology of a back propagation neural network*

ward. The Adaptive Resonance Theory (ART) model is a complex trained feedback paradigm developed by Stephen Grossberg and Gail Carpenter of the Center for Adaptive Systems at Boston University. ART is considered by some to be very powerful, but the number of patterns that can be stored is limited to exactly the number of nodes in the storage layer. No production applications have been published to date; ART is presently considered a research tool.

Feed-Forward Topologies

The second division of neural networks is the feed-forward category. The earliest neural network models were linear feed-forward. In 1972, two simultaneous papers independently proposed the same model for an associative memory, the linear associator. J.A. Anderson, a neurophysiologist, and Teuvo Kohonen, an electrical engineer, were not aware of each other's work.

The linear associator uses the simple Hebb's Rule. The only case where association is perfect when simple Hebbian learning is used is when the input patterns are orthogonal. This puts an upper limit on the number of patterns that can be stored. The system will work very well for random patterns if the maximum number of patterns to be stored is 10 – 20 percent of the number of neurons. If the input patterns are not orthogonal, there will be interference among them; fewer patterns can be stored and correctly retrieved. One of the predictions of the linear associator is interference between nonorthogonal patterns. Much of Kohonen's book, *Self-Organization and Associative Memory* (Springer-Verlag, 1984) is concerned with correcting the errors caused by interference.

The nonlinear feed-forward models are the most commonly used today. Feed-forward networks, for historical reasons, are less often considered to be associative memories than the feedback networks, even though they can provide exactly the same functionality. It can be shown mathematically that any feedback network has an equivalent feed-forward network that performs the same task.

Types of Learning Algorithms

There are two main types of training algorithms: Supervised and unsupervised. Supervised learning is the most elementary form of adaptation. It requires an a priori knowledge of what the result should be. During training, the network's output is compared to the ideal response, and any error is used to correct the network. Learning occurs as a result of changes to the weights to reduce the errors as the network gains experience. For one-layer networks this is easily accomplished by monitoring each neuron individually. In multi-layer networks, supervised learning is more difficult due to the correction of the hidden layers. Unsupervised learning differs in that it does not have specific corrections made by comparison to ideal results. Supervised and unsupervised learning are methods which are used exclusively of each other.

The supervised back propagation model is the most commonly implemented paradigm today because it is the best general-purpose model and probably the best at generalization. (This model is used by the "BrainMaker" software from California Scientific Software.) Back propagation is a multi-layer feed-forward network that uses the Generalized Delta Rule.

By 1985, back propagation had been simultaneously discovered by three groups of people: D.E. Rumelhart, G.E. Hinton, R.J. Williams; Y. Le Cun; and D. Parker. Back propagation is the canonical feed-forward network where an error signal is fed back through the network, altering weights as it goes, in order to prevent the same error from happening again. (See Figure 4.)

The error on an output neuron, i, for a particular pattern, p, is defined as $E_{pi} = (T_{pi} - O_{pi})$ where T is the training (desired) pattern and O is the actual output. The total error on pattern p, E_p, is the sum of the errors on all the output neurons for pattern p. The total error, E, for all patterns is the sum of the errors on each pattern over all p. The simplest method for finding the minimum of E is known as "gradient descent." It involves moving a small step down the local gradient of the scalar field. This is directly analogous to a skier always moving down hill through the mountains until he hits the bottom.

Back propagation is useful because it provides a mathematical explanation for the dynamics of the learning process. It is also very consistent and reliable in the kinds of applications that can currently be built. The biggest limi-

tation is the size of the network. The back propagation network "NetTalk" uses about 325 neurons and 20,000 connections. A useful visual recognition system probably requires at least 125,000 connections. Currently available commercial systems provide anywhere from a few neurons and connections to 1 million neurons and 1.5 million connections, for anywhere from $200 to $25,000.

A popular unsupervised feed-forward model is the Kohonen model. The basic system is a one- or two-dimensional array of threshold-type logic units with short-range lateral connections between neighboring neurons. The system modifies itself so that nearby neurons respond similarly. The neurons compete in a modified winner-take-all manner. The neuron whose weight vector generates the largest dot product with the input vector is the winner and is permitted to output. In this model not only the weights of the winner but also those of its nearest neighbors (in the physical sense) are adjusted.

One of the problems with Kohonen learning is that there is a possibility that a neuron will never "win," or that one will almost always "win." The weight vectors get stuck in isolated regions. One way to prevent the weight vectors from getting stuck is to start off with all the weight vectors equal. The network is first fed fractional amounts of the patterns. The inputs are then slowly built up to the full input patterns. This method, called "convex combination," works well but it slows down learning. Another preventative method is to add noise to the data, which makes the probability density function positive everywhere. The probability density function is a real-valued function that gives the probability that a random variable has values in the set. This method works, but it is even slower than convex combination. Another approach is to give the neurons a "conscience"; if the neurons realize that they are winning a lot, they will step out of the competition for a while.

A special case of the feed-forward model is the Neocognitron. The original model was unsupervised, but a more recent model (1983) uses a teacher. The multi-layer (seven- or nine-layer) system assumes that the builder of the network knows roughly what kind of result is wanted. All the neurons are of analog type; the inputs and outputs take nonnegative values proportional to the instantaneous firing frequencies of actual biological neurons. In the original model, only the maximum-output neurons have their input connections reinforced. It uses a variation of the Hebbian Rule. After learning is completed, the final Neocognitron system is capable of recognizing handwritten numerals presented in any visual field location, even with considerable distortion. Drawbacks of the Neocognitron are that it is highly specialized and requires a large number of neurons and connections.

Conclusion

Neural networks are capable of some impressive things but they are also limited, primarily by the size of the network and the complexity of the problem. They are especially good at association and generalization, but poor at precise computations and logic. Some models are able to generalize better than others, some are good at association.

With more than 40 functioning models to choose from, it is important to know which models have had the most success and to understand their similarities and differences. Currently, back propagation is the most popular model. Several others are discussed in detail in this issue, each has it own merits.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

Implementing the Rhealstone Real-Time Benchmark

Where a proposal's rubber meets the real-time road

Rabindra P. Kar

In February 1989, the late Kent Porter and I proposed a set of benchmarking operations for real-time multitasking systems (see "Rhealstone: A Real-Time Benchmarking Proposal," *DDJ*, February 1989). That article generated a lot of interest and many valuable suggestions from *DDJ* readers, as well as others in the real-time software community. The reader response made it possible for us to refine and clarify several important aspects of the original benchmark proposal. I am very grateful to those of you who shared your insights with us.

This article presents the refined definition of the Rhealstone benchmark. It also contains a suite of C programs that implement the benchmark under iRMX, a real-time operating system from Intel. Refer to the original proposal for the rationale behind proposing Rhealstones in the first place, and for background information on the real-time multitasking operations that comprise it.

First, I'll give a quick summary of what the Rhealstone benchmark is and what it seeks to measure. The benchmark identifies the execution times (or time delays) associated with six operations that are vital indicators of real-time multitasking system performance. These six operations are named "Rhealstone components." When a real-time system is benchmarked, each of these components is measured separately. The empirical results are combined into a single figure of merit (Rhealstones per time unit). There are two ways of calculating the Rhealstone number: One of them generic and one of them appropriately weighted for a particular type of application (an application-specific Rhealstone).

Rhealstones are intended to provide a standard of comparison between real-time computers across the industry. Hence, their specification is: a. Independent of the features found in any CPU; b. Independent of any computer bus architecture; c. Independent of the features or primitives of any operating system or kernel (collectively referred to hereafter as real-time executives).

The C language implementation of the benchmarks is, of course, specific to an operating system (iRMX in this case). However, the OS-specific part of the code is confined to a few system

Robin is a senior engineer with the Intel Systems Group and can be reached at 5200 N.E. Elam Young Parkway, Hillsboro, OR 97124-6497.

Reader's Rhealstone Recommendations

When the original Rhealstone proposal was put forth more than a year ago, *DDJ* solicited comments, suggestions, and recommendations from readers. In addition to the individuals Robin has acknowledged in this article, the following readers contributed comments. If any contributors were left off this list, it is unintentional and we apologize — please let us know who you are. The version of the benchmark presented in this article does not necessarily represent Rhealstone as it will look in years to come. We look forward to your suggestions and recommendations for this version too.

— Eds.

Mark Smotherman, Clemson University; Glenn Yeager, Applied Integration Management Corp.; Colburn L. Norton, Baytown, Texas; Gary Osborne, Apricot Computers; Jim D. Hart, Papillion, Nebraska; John Morgan, Bellingham, Washinton; G. Bruce Lott, Real Time Systems; Michael S. Sossi, Leo Burnett USA; Rudi Borth, Stratford, Ontario; Carol Sigda, Industrial Programming Inc.; Phil Daley; Hillsboro, New Hampshire; Tim Olson, Advanced Micro Devices.

(continued from page 46)
calls (to create tasks, put them to sleep, accept/relinquish semaphores, and so on) that are found in almost every multitasking executive. The C benchmark source is easily portable to most other executives if the iRMX system calls in it are replaced by the equivalent system calls of the target executive.

When a computer is used in a real-time-control situation, the solution usually fits the model described in Figure 1.

"Real-time computer system" refers jointly to the CPU and system software (OS, kernel, or combination thereof) that provide a base execution vehicle for the application software. The Rhealstone benchmark helps the real-time solution designer choose the highest-performance real-time computer available as the execution base for the project. Rhealstones are not designed to measure how good the complete solution is, and they may not be an appropriate measure for the end user.

Rhealstone Components

The specifications for the six real-time operations (Rhealstone components) that comprise the benchmark are detailed in the following section. For a graphical specification of each component, refer to Figures 2 through 7. Each component's specification is followed by a paragraph (or two) describing the C benchmark program used to measure the Rhealstone component on iRMX II. It is important to realize that the verbal and graphical specifications, not the C programs, are the essential core of the benchmark. It is entirely possible to obtain more accurate Rhealstone component values by using different algorithms or programming languages or special performance-analysis hardware.

The task-switch time (see Figure 2) is the average time to switch between two active tasks of equal priority. The tasks should be independent — that is, there should not be contention between them for hardware resources, semaphores, and so on. Task switching must be achieved synchronously (without preemption) — for example, when the running task puts itself to sleep or when the executive implements a round-robin scheduling algorithm for equal-priority tasks.

Listing One (tswit.c), page 100, is a program to measure task-switch time. The code of *task1* and *task2* is identical and very simple: A loop in which *rqsleep* gets called in every iteration. The iRMX system call *rqsleep(sleep_time, return_code_pointer)* lets a task put itself to sleep (suspend execution) for *sleep_time* system clock ticks (in iRMX, the default clock tick is 10 milliseconds long). If *sleep_time* is 0, the task is not necessarily put to sleep; rather, iRMX will switch execution to any other equal-priority task that is ready to run (which is why this program calls *rqsleep*).

The *return_code_pointer* is the last parameter of every iRMX system call. It points to an unsigned variable in which iRMX places the status of the call. If the status returned is 0, the call has been executed correctly; if this is not so, the status returned is an error or warning code indicating why the call did not execute as it should have.

The call *rqgettime (return_code_pointer)* returns the number of seconds that have elapsed since a fixed point in time. It is called twice, to measure the elapsed time (in seconds) between any two points in a program.

The call *rqcreatetask (priority_level, start_address, data_seg, stack_pointer, stack_size, task_flags, return_code_pointer)* creates a new task, as its name suggests. The first parameter sets the new task's priority between 0 (highest priority) and 255 (lowest priority). The other parameters are either self-explanatory or iRMX programming details. This call returns a *task_token* that becomes iRMX's identifier for the task. Consequently, the *rqdeletetask(task_token, return_code_pointer)* call uses *task_token* to identify which task is to be deleted. If *task_token* is NULL or 0, the task deletes itself.

The *rqgetpriority(task_token, return_code_pointer)* call lets a task find out the priority level of any existing task in the system. If *task_token* is NULL or 0, the priority of the calling task itself is returned.

Finally, the call *rqsetpriority(task_token, priority_level, return_code_pointer)* lets a task dynamically change its own or any other task's priority. I've used it to set the main task's priority to be lower than those of *task1* and *task2* so those tasks can run to completion without interference from the main program.

The preemption time (see Figure 3) is the average time for a high-priority task to preempt a running low-priority

Real-time computer system	+	Real-time application code	+	Human-machine and/or machine-machine interface(s)	=	Real-time solution

Figure 1: *Typical real-time control model*

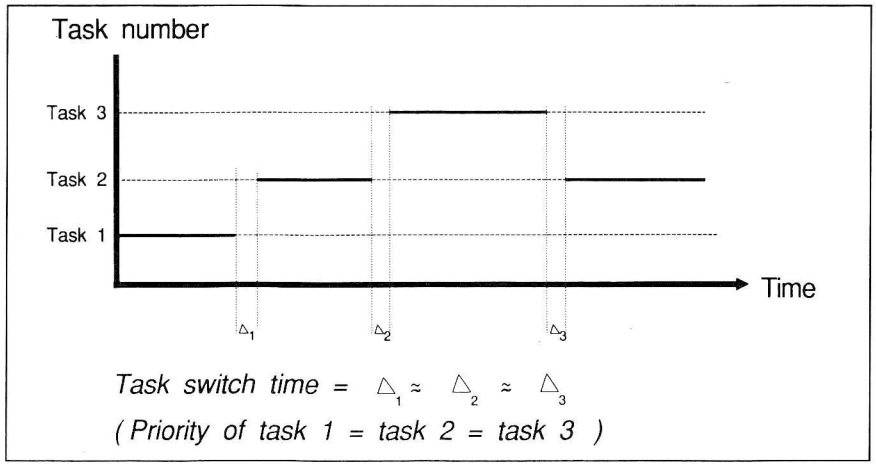


Figure 2: *Task-switch time*

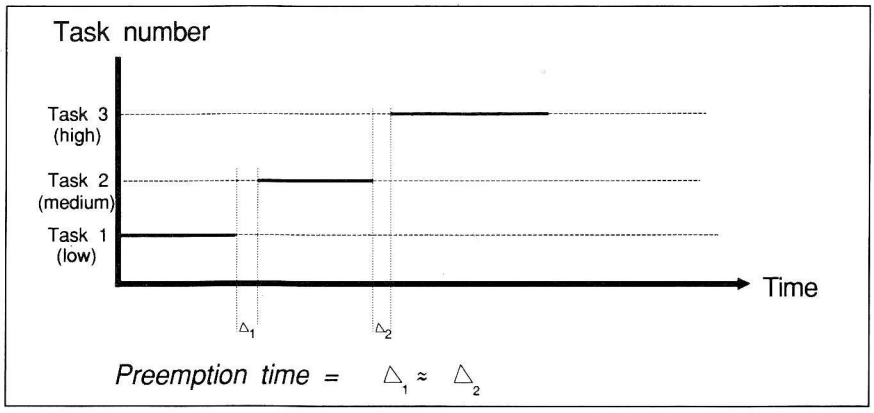


Figure 3: *Preemption time*

(continued from page 48)
task. Preemption usually occurs when the high-priority task goes from a suspended or sleeping state to a ready state because either the high-priority task wakes up from a previously initiated sleep, or some event or signal that the task was waiting for is recognized. The first case will likely yield a lower preemption time on most systems, and that value is acceptable for the Rhealstone benchmark.

Listing Two (preempt.c), page 100, measures preemption time in an iRMX II system. *task1* (lower priority) merely sits in a delay loop, waiting to be preempted. *task2* loops on an *rqsleep* call. Every time it calls *rqsleep* a (non-preemptive) switch to *task1* takes place. When one sleep period is over, *task2* wakes up and preempts *task1*.

Interrupt latency (see Figure 4) is the average delay between the CPU's receipt of an interrupt request and the execution of the first application-specific instruction in an interrupt-service routine. The time required to execute machine instructions that save the CPU's context (CPU's and coprocessor's data registers, mode registers, and so on) are part of the interrupt latency.

As just defined, interrupt latency reflects only the delay introduced by the executive and the CPU itself. It does not include delays that occur on the system bus or electrical delays in control circuitry external to the computer system because control circuitry is usually specific to the application.

Interrupt latency can be measured under iRMX II on a PC/AT-compatible computer using ltncy.c (Listing Three, page 100) and latch.asm (Listing Four, page 101). The benchmarking technique used here is different from the one I've used for the other Rhealstone components because interrupt latency is about an order of magnitude smaller than those components (under iRMX), hence the need for greater accuracy in measurement. The first difference is that this benchmark involves a C and an assembler program. Secondly, the latency is measured by reading the 8254 timer chip directly (bypassing iRMX). Of course, this makes the benchmark hardware-dependent, which is the major disadvantage of this technique.

The code in ltncy.c sets up a new interrupt vector (pointing to the assembler code in latch.asm) with the *rqsetinterrupt(encoded_int_level, int_task_flag, int_handler, int_handler_data_seg, return_code_pointer)* call, reads the timer chip, and simulates a hardware interrupt with an *INT (causeinterrupt)* instruction in software. The processor vectors to latch.asm, which saves context (by pushing CPU registers on the stack) and reads the timer again.

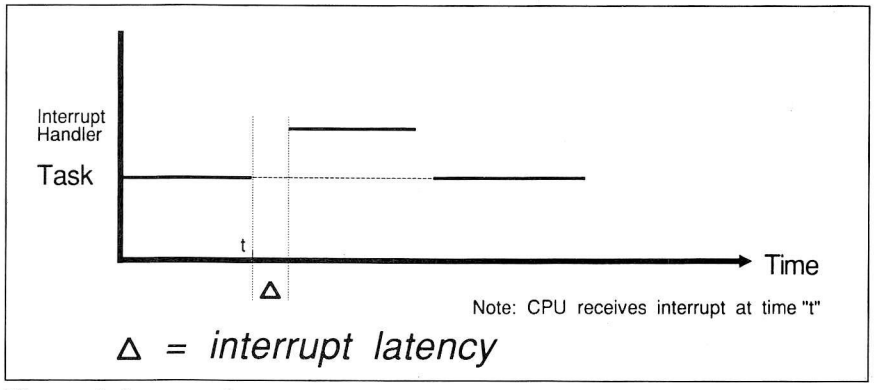


Figure 4: *Interrupt latency*

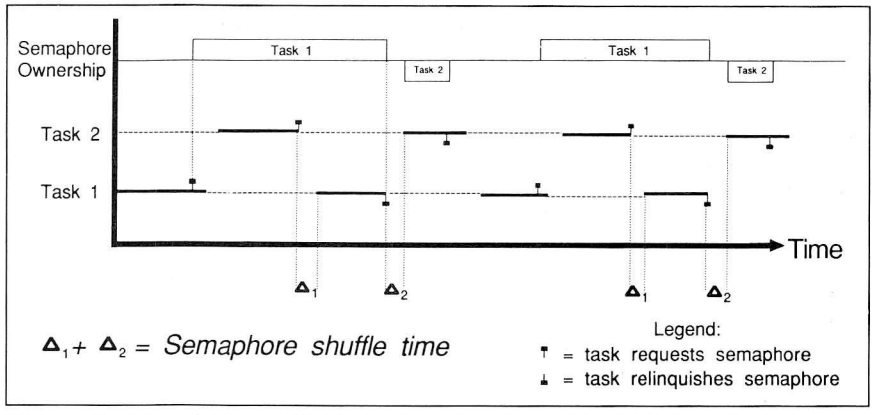


Figure 5: *Semaphore-shuffle time*

The difference in the two timer count values is used to calculate interrupt latency in microseconds.

The call *rqresetinterrupt(encoded_interrupt_level, return_code_pointer)* merely restores the interrupt vector to its default value (before the *rqsetinterrupt* call).

Semaphore-shuffle time is the delay/overhead, within the executive, before a task acquires a semaphore that is in the possession of another task when the acquisition request is made. Figure 5 illustrates what is being measured. *task2* requests a semaphore that *task1* owns. Semaphore-shuffle time is the delay within the executive (excluding the run time of *task1* before it relinquishes the semaphore) between *task2*'s request and its receipt of the semaphore.

The objective here is to measure overhead when a semaphore is used to implement mutual exclusion. Many real-time applications involve multiple tasks needing access to the same resource. Semaphore-based mutual exclusion is a convenient way to ensure that the resource is not interrupted by a second task before it finishes an operation started by the first task.

Listing Five (semshuf.c, page 101) shows a program that measures semaphore-shuffle time. *task1* and *task2* are first executed a fixed number of times, without any semaphore-related calls. Then these tasks are executed again, the same number of times, with a semaphore being shuffled between them at every iteration. The difference in execution time with and without semaphore shuffling is the overhead within the executive. This program uses the following three iRMX system calls associated with semaphores.

The Rhealstone benchmark identifies the execution times (or time delays) associated with six operations that are vital indicators of real-time multitasking system performance

The call *sem_token = rqcreatesemaphore(initial_value, max_value, queuing_method, return_code_pointer)* creates a new counting semaphore. A counting semaphore can be incremented up to the *max_value* parameter. Because this program sets *max_value* to 1, it is using a simple binary semaphore. The parameter *queuing_method* is set to 0 for a FIFO queuing scheme when more than one task is waiting on the semaphore; a value of 1 would make it a priority-based queue.

The *rqsendunits(sem_token, units_sent, return_code_pointer)* call increments the value of the semaphore. In this program, the calling task uses it to relinquish the semaphore.

Finally, the call *remaining_units = rqreceiveunits(sem_token, units_requested, max_wait_time, return_code_pointer)* decrements the value of the semaphore. The task makes this call to acquire the semaphore if available. The *max_wait_time* parameter specifies (in system clock ticks) the period it is willing to wait for the semaphore. A value of 0xffff means "wait forever."

Deadlock-break time is the average time to break a deadlock caused when a high-priority task preempts a low-priority task that is holding a resource the high-priority task needs.

Figure 6 illustrates how a deadlock

(continued from page 51)
situation occurs. Task 1 gains control of the resource and is preempted by medium-priority task 2. High-priority task 3 preempts task 2 and requests the resource from the executive at some point. Because task 1 still holds the resource, task 3 is now blocked. At this point, an unsophisticated executive would resume task 2. Task 2 knows nothing about the critical resource and may block higher-priority task 1 indefinitely (a deadlock situation!). A good real-time executive would not resume task 2 here (below the ? in Figure 6). To avoid the deadlock, the executive might temporarily raise task 1's priority to the same level as task 3's, until it relinquishes the resource. In any case, the benchmark measures the delay between task 3's request and its acquisition of the resource, excluding task 1's run time before it relinquishes the resource.

The program deadbrk.c (Listing Six, page 102) measures deadlock-break time in iRMX II. The algorithm is similar to the semaphore-shuffle benchmark. Three tasks of different priorities are executed a fixed number of times without competing for a critical resource. The same tasks are executed again, but this time *task1* and *task3* both access the same resource, with a potential deadlock situation occurring in each iteration. The difference in total execution time between the two cases is a measure of the deadlock break time.

Access to a critical resource is guarded by an iRMX object called a "region." The region is created by the *region_token = rqcreateregion(queuing_method, return_code_pointer)* system call. The *queuing_method* parameter has the same function here as in the *rqcreatesemaphore* call. Because only one task should access the critical resource at a time (mutually exclusive access), each task waits at the *rqreceivecontrol(region_token, return_code_pointer)* call until the region is free. The task must relinquish control with the *rqsendcontrol (return_code_pointer)* call when it has finished with the resource.

Intertask message latency is the latency/delay within the executive when a nonzero-length data message is sent from one task to another (see Figure 7). To best measure intertask message latency, the sending task should stop executing immediately after sending the message and the receiving task should be suspended while waiting for it.

The message-passing mechanism must obey two important conditions: First, the intertask message-passing link must be established at run time. (Passing data in a predefined memory area,

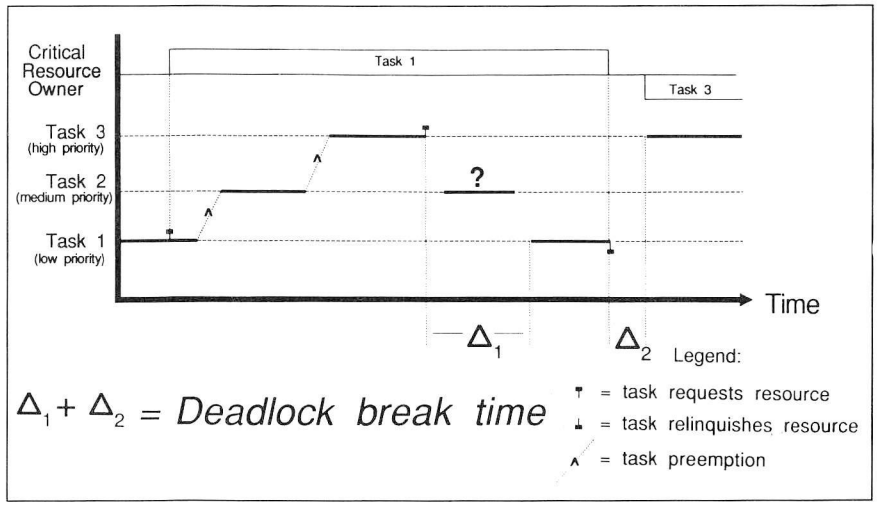


Figure 6: *Deadlock-break time*

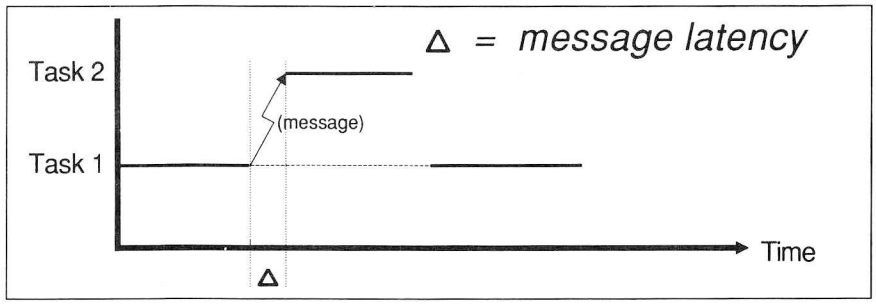


Figure 7: *Intertask message latency*

such as a global variable, is not permitted.) Second, if multiple messages are sent on the same link, the sending task must not be allowed to overwrite an old message with a new one before the receiving task gets a chance to read it. Multitasking executives typically offer mechanisms such as pipes, queues, and stream files for intertask data communications.

The program it_msg.c (Listing Seven, page 104) measures message latency in iRMX II. Data messages are passed between tasks in an iRMX mailbox. The mailbox is created by the *mailbox_token = rqcreatemailbox(type_flags, return_cod_pointer)* system call. A task sends a data message to another task by calling *rqsenddata(mailbox_token, message_pointer, message_length, return_code_pointer)*.

The receiving task calls *message_length = rqreceivedata(mailbox_token, receive_buffer, max_wait_time, return_code_pointer)* to receive the message, if available. If not, the task is made to wait until *max_wait_time* clock ticks have elapsed (if this parameter is 0xffff, the receiving task is willing to wait as long as is necessary).

Note: This Rhealstone component is a modification of "datagram throughput," which was proposed in the original article. The specification of intertask message latency partly reflects reader input (see acknowledgment 1).

Computing a Rhealstone Performance Number

Measurement of all six Rhealstone components yields a set of time values (in the tens of microseconds to milliseconds range, for most PCs). Although the individual measurements are of significance by themselves, it is useful to combine them into a single real-time figure of merit, so overall comparisons between real-time computers can be made. To get a single Rhealstone performance number, the following computational steps are necessary:

1. All Rhealstone component time values should be expressed in the same unit (seconds).
2. The arithmetic mean of the components must be computed.
3. The mean (from step 2) must be arithmetically inverted to obtain a consolidated real-time figure of merit, in Rhealstones/second.

Given the following set of measured values:

task-switch time = t1 seconds
preemption time = t2 seconds
interrupt latency = t3 seconds
semaphore-shuffle time = t4 seconds
deadlock-break time = t5 seconds
intertask message latency = t6 seconds

the arithmetic average of the Rhealstone components is:

$$t' = (t1 + t2 + t3 \ldots\ldots + t6) / 6$$

and the system's consolidated real-time performance number is:

$$1/t' \text{ Rhealstones/second}$$

Application-Specific Rhealstones

The operational definition of Rhealstones, in the previous section, is generic for any application that may be executed on a real-time computer. It treats all the Rhealstone components as equally important parameters of real-time performance. Generic benchmarks are useful when evaluating real-time system performance without a particular application in mind.

When a real-time computer is "dedicated" to a type of application, the Rhealstone figure can be computed in a way that is appropriate to it. This performance figure is called an "application-specific Rhealstone." It gives unequal weight to different Rhealstone components because the application's performance is not influenced by all

of them equally. For example, the application may be heavily interrupt-driven, and the software may not use semaphores at all. The application's designer can compute the application-specific number if he/she knows (or can estimate) the relative frequency of different Rhealstone components in the application.

The steps for computing application-specific Rhealstones are as follows:

1. Measure the individual components (t1 to t6) as before.
2. Estimate the relative frequency of each Rhealstone component's occurrence when the application is executed, and assign nonnegative real coefficients (n1 to n6) proportional to the frequencies. For example, if interrupts occur five times more often than task switches do, and semaphores and intertask communication are not used in the application code, the value of n3 should be three times the value of n1, and n4 and n6 should be set to 0.
3. Compute a weighted average of the Rhealstone components:

```
t' = (n1*t1 + n2*t2 + n3*t3 + . . . . +
      n6*t6) / (n1 + n2 + n3 . . . + n6)
```

4. Invert the average to get the result 1/t' application-specific Rhealstones/second.

The procedure for computing generic and application-specific Rhealstones outlined in this article is different from that specified in the original proposal. The original procedure specified that each Rhealstone component should be arithmetically inverted separately and the average taken thereafter. I am grateful to the many readers who wrote to point out that, with the previous procedure, a computer system with high performance in one or two components and bad performance in the others would outshine one with moderately good performance in all categories.

The revised algorithm specifies that the components be averaged first and then arithmetically inverted (see acknowledgment 2). This algorithm ensures that if a real-time system shows bad performance in even one category, its overall score will suffer badly. This is intentional, because to guarantee quick response time, moderately good performance is needed in all Rhealstone categories. In other words, a real-time system that takes several seconds to respond to an interrupt will have a low Rhealstone rating, even if it delivers microsecond-range performance when switching context or exchanging semaphores.

Acknowledgment

1. Robert Wilson, GE Huntsville, Alabama, and Ketan Sampat, Intel Hillsboro, Oregon, for their input on measurement of inter-task communication performance.
2. Marco Pellegrino, Siemens AG Munich, Federal Republic of Germany, for suggestions on computation of Rhealstone performance number.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 100.)

Vote for your favorite feature/article.
Circle Reader Service **No. 4.**

Bounding Box Data Compression

An optimized font data compression method for fast screen I/O environments

Glenn Searfoss

The optimal data compression method for a given situation is determined by the type of data to be used and its intended application. To date, a variety of methods have been used to effect a balance between compression and display speed of bit-mapped font data.

One school of thought maintains that only noncompressed font data be used. While this data can be displayed rapidly, the amount of data storage required for fonts can be prohibitive, particularly when used with higher display resolutions and colors.

Another school stresses maximum compression of font data at all times. In this situation, each time a character is accessed, the cell data must be reconstituted. This saves data storage space but sacrifices access speed during screen display.

A third approach attempts to incorporate both aspects. Here font data remains compressed until accessed, at which point all data is uncompressed. A font may then be used at optimal speed. As additional fonts are uncompressed, the data storage limitation of the first method is encountered. This limitation can be ameliorated by each caching system in which only the most recently used characters are available in uncompressed form (at the expense of additional code complexity).

Because none of these methods are completely satisfactory for font screen display, it is important to develop a better balance between efficient access and efficient storage.

The terms "fast-access" or "on-the-fly" have been used to describe bit-mapped font data optimized for fast screen I/O. It is ultimately desirable in this environment to achieve maximum data compression while maintaining or increasing data access speed. To this end, it is essential to understand certain restrictions inherent with the graphics display of bit-mapped font data.

First, there must be minimal calculation of font data. Vector format and highly compressed bit-mapped fonts both require recalculation of character data at display time. This greatly restricts their usefulness in a fast-access environment. Speed optimization occurs when there is little or no data compression and reconstruction.

Second, character size must relate to the display resolution. As screen display resolution increases, larger characters are required to maintain the same relative size and quality as characters used on lower-resolution displays.

Third, font data storage requirements must not impinge upon code space. As bit-mapped fonts increase in size, their data storage needs quickly reach mammoth proportions. Realistically, some form of data compression is required for a program and several large font sets to coexist in memory.

Fourth, the amount of code necessary must be functional, effective, and preferably compact. Coding requirements are reduced with minimal data compression and decompression.

The data compression method best suited for this application will achieve a dynamic balance between these four criteria. The Bounding Box method of data compression is such a method.

To illustrate its effectiveness in this situation, a comparison with a commonly used method of data compression, run length bit encoding (RLE), is useful. This comparison is useful even though the two approaches are not mutually exclusive: One can store run length encoded characters in the Bounding Box format.

A valid comparison depends upon establishing a common reference point. Because noncompressed data is the basic requirement for both compression schemes, a brief outline of this font data format may be useful.

Noncompressed fonts are used as is. The font data is comprised of header information and complete character cell data. The header may detail as many character formatting aspects as desired.

A typical header for standard noncompressed font data is shown in Listing One, page 108. In this case, I'm using the C language data structure of Data Transforms' Fontrix (Font1) Format.

Each font has a header that defines the font and points to the character bitmaps. Immediately following the font header is the character bitmap data. Character bitmaps are stored as scanrects in scanline order from top to bottom. Each scanline is stored byte wise left to right, left justified, and rounded to byte length. Each byte is stored 8 bits per byte where MSB is the leftmost pixel. Using this as a standard font header to describe Figure 1, we can proceed to specifics concerning methods of compressing this data.

Glenn works at Data Transforms Inc., and can be reached at 616 Washington Street, Denver, CO 80203.

(continued from page 56)

Bounding Box Compression

The Bounding Box Compression Method (see Figure 2) involves outlining a cell's "bit-on" character data with the smallest box possible. Coordinates that position the "box" relative to the original character cell are saved in the font header. This compression method is automatic within the Data Transforms Font Editor when a font over 32 × 32 pixels in size is saved. A sample header for a font compressed using the Bounding Box method is shown in Listing Two, page 108 (again using Data Transforms' Fontrix [Font2] Format).

The data listed in the *struct font2* portion of the (Font2) font structure above is defined as follows:

- The font header is the same as described in the *struct font1head* of the noncompressed font data format; font cell segments are an array of segment pointers to characters kept as offsets from the beginning of the font file. For example, if an array value for a character = 10, then the starting address of a character's "bounding box" data = (the start of file address + *size of [struct font2])* + (10 × 16), where 1 segment = 16 bytes.
- The horizontal size is the actual width in bits of the bounding box.
- The horizontal offset is the distance in bits from the left edge of the cell to the upper lefthand corner of the bounding box.
- Horizontal bytes refers to the actual size in bytes of the interior of the bounding box.
- The vertical size is the actual height in bits of the bounding box.
- The vertical offset is the distance in bits from the top edge of the cell to the upper edge of the bounding box.

The character data is never compressed; rather, the empty space outside the "bounding box" is discarded. The analogy of a shrinkwrap bag can be used to illustrate.

Imagine a character cell placed within a shrink-to-fit bag. The noncompressed cell is the unshrunk bag. Now shrink the bag until all edges contact the outer limits of bit-on data, forming a rectangular bounding box. For some characters — such as a lowercase "i" — this correlates to a major amount of shrinkage, and the shrinkage correlates to saved data space, hence, compression. An uppercase "W" may fill most of a cell. In this instance minimal shrinkage will occur, with little or no saved data space. However, the overall net savings in data space for an entire font set will be great because few characters fill an entire cell (see Figures 2, 3, and 4).

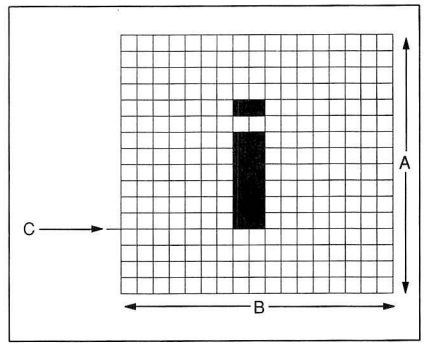


Figure 1: *A noncompressed character cell.* ***A.*** *Vertical cell size 16 pixels.* ***B.*** *Horizontal cell size 17 pixels.* ***C.*** *Baseline character cell 12 pixels*

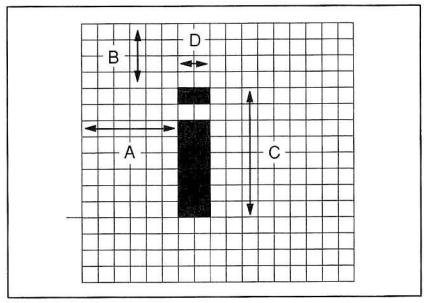


Figure 2: *A character cell compressed using the Bounding Box method. The data listed is saved for each character when the font was created and stored in a look-up table in the font header.* ***A.**** *Horizontal offset 6 pixels.* ***B.*** *Vertical offset 4 pixels.* ***C.*** *Vertical size of bounding box 8 pixels.* ***D.*** *Horizontal size of bounding box 2 pixels*

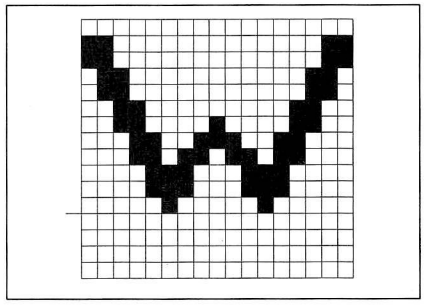


Figure 3: *Noncompressed character cell. Vertical cell size 16 pixels. Horizontal cell size 17 pixels. Baseline character cell 12 pixels*

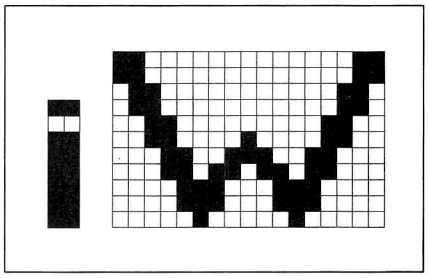


Figure 4: *Actual data saved when a character cell is compressed using the Bounding Box method*

(continued from page 58)

Run Length Bit Encoding

A RLE font would possess a header similar to the font header described in the "struct font1head" of the noncompressed font data format. The main differences between RLE and the Bounding Box lies in the method of character data storage and how code points to it.

In a simple case of run length bit encoding, bit mapped data is compressed by reading each scanline of a character cell, grouping adjacent bit-on or bit-off data and saving the information as pairs of ASCII digits. For example, by using RLE the marked scanline in Figure 5 could be compressed as follows:

- 0x06 (6 identical bits in a row), 0x00 (those bits are bit-off data)
- 0x02 (2 identical bits in a row), 0x01 (those bits are bit-on data)
- 0x09 (9 identical bits in a row), 0x00 (those bits are bit-off data)

Having established the basic structure of the two methods, it now remains to compare their differences in access speed, compression efficiency, and coding requirements.

Speed of Compressed Data Access

When a Bounding Box compressed character is used, there is no run-time penalty during screen display. A character cell is not reconstituted. Rather, the character data is used as is and positioned relative to the original cell size information.

Fonts compressed with run length bit encoding pay a run-time penalty during screen display. If a font remains compressed while being accessed, each time a character is displayed to the screen, the entire character cell must be reconstructed. In a case like this, you'll be watching the clock.

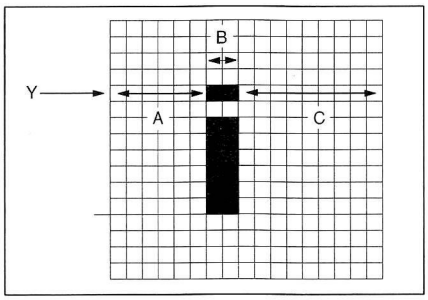


Figure 5: *Run length bit encoding for scan line Y. Each portion of data is compressed into a pair of ASCII digits.* ***A.*** *0x06 0x00* ***B.*** *0x02 0x01* ***C.*** *0x09 0x00*

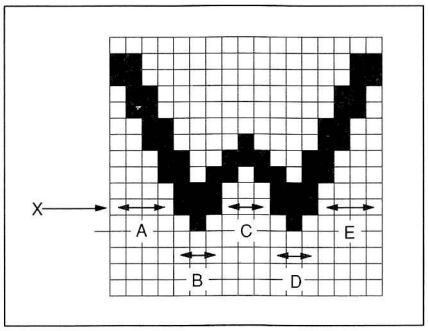


Figure 6: *Run length bit encoding for scan line X. Each portion of data is compressed into a pair of ASCII digits. The more data dispersal on a scan line, the more detailed this method of compression becomes.* ***A.*** *0x04 0x00* ***B.*** *0x03 0x01* ***C.*** *0x03 0x00* ***D.*** *0x03 0x01 E. 0x04 0x00*

Compression Efficiency

The Bounding Box routine can run at up to 90 percent of the overall compression efficiency of more computationally expensive compression algorithms. The actual character bits-on data is never compressed.

For this reason, the Bounding Box approach is more properly called a technique or a character storage format, rather than an algorithm. Information regarding the bounding box (size, [x,y] position within the original cell, and so on) is kept for each character in a lookup table in the font header (see Figures 2 and 4).

Using this compression method on the character cell in Figure 1 (lowercase i), the net gain in savings is 94 percent of the original character cell size. For Figure 3 (uppercase W), the net savings is 31 percent of the original character cell size. The percentage in savings correlates to the discarded zero (bit off) data outside the bounding box (see Figure 4).

The run length bit encoding (simple case) compression method can run at up to 98 percent of overall compression efficiency, but will be computationally expensive. Using this compression method on the character cell in Figure 1 (lowercase i), the net gain in savings is 78 percent of the original character cell size. For Figure 3 (uppercase W), the net savings is 57 percent of the original character cell size (see Figures 5 and 6). Table 1 provides a complete RLE table for Figure 5. Read

(continued from page 60)
each scan line left to right, top to bottom, numbered 0 to 15. Table 2 is the complete RLE table for Figure 6. Again, read each scan line left to right, top to bottom, 0 to 15.

Coding Requirements

The Bounding Box method is invoked at a time when display speed is not an issue — during font creation. All the data needed to use a character is saved in a lookup table in the font header at this time. Because the actual data within the bounding box is not compressed, no coding is required to reconstruct a character cell. Code need only index into the data and index the screen pixel position.

Run length bit encoding requires code to do more than access font data. It must also handle the peculiarities involved with compressing and uncompressing data. Font characters that remain compressed while being used must be reconstructed each time they are accessed. Font data that is reconstituted once and thereafter accessed as noncompressed data, can instigate a data storage conflict. As additional fonts are uncompressed, their combined data storage requirements begin to compete with code and operational program space.

The Nitty Gritty

The Bounding Box method is most effective when the data to be compressed is bit-mapped data; all bit-on data is concentrated and confined in one area, as is often the case with alphanumeric character fonts; the ratio of bit-off to bit-on data is high; fast usage of compressed data is of most importance (for example, screen I/O, graphics printing, and so on); and it is preferred that font data remain compressed when being accessed.

```
 0] 0x17 0x00
 1] 0x17 0x00
 2] 0x17 0x00
 3] 0x17 0x00
 4] 0x06 0x00 0x02 0x01 0x09 0x00
 5] 0x17 0x00
 6] 0x06 0x00 0x02 0x01 0x09 0x00
 7] 0x06 0x00 0x02 0x01 0x09 0x00
 8] 0x06 0x00 0x02 0x01 0x09 0x00
 9] 0x06 0x00 0x02 0x01 0x09 0x00
10] 0x06 0x00 0x02 0x01 0x09 0x00
11] 0x17 0x00
12] 0x17 0x00
13] 0x17 0x00
14] 0x17 0x00
15] 0x17 0x00
```

Table 1: *RLE table for Figure 5. Reading each scan line left to right, top to bottom, and numbered 0– 15*

```
 0] 0x17 0x00
 1] 0x02 0x01 0x13 0x00 0x02 0x01
 2] 0x02 0x01 0x13 0x00 0x02 0x01
 3] 0x01 0x00 0x02 0x01 0x13 0x00 0x02 0x01 0x01 0x00
 4] 0x01 0x00 0x02 0x01 0x13 0x00 0x02 0x01 0x01 0x00
 5] 0x02 0x00 0x02 0x01 0x13 0x00 0x02 0x01 0x02 0x00
 6] 0x02 0x00 0x02 0x01 0x04 0x00 0x01 0x01 0x04 0x00 0x02 0x01 0x02 0x00
 7] 0x03 0x00 0x02 0x01 0x02 0x00 0x03 0x01 0x02 0x00 0x02 0x01 0x03 0x00
 8] 0x03 0x00 0x02 0x01 0x01 0x00 0x02 0x01 0x01 0x00 0x02 0x01 0x01 0x00 0x02 0x01 0x03 0x00
 9] 0x04 0x00 0x03 0x01 0x03 0x00 0x03 0x01 0x03 0x00
10] 0x04 0x00 0x03 0x01 0x03 0x00 0c03 0x01 0x04 0x00
11] 0x05 0x00 0x01 0x01 0x05 0x00 0x01 0x01 0x05 0x00
12] 0x17 0x00
13] 0x17 0x00
14] 0x17 0x00
15] 0x17 0x00
```

Table 2: *RLE table for Figure 6. Reading each scan line left to right, top to bottom, and numbered 0– 15*

(continued from page 62)

Run length bit encoding is most effective when all bit-on data is widely dispersed or present at a majority of the cell edges; the ratio of bit-off to bit-on data is low; and the usage of data storage must be maximized or is of higher priority.

RLE and the Bounding Box are two good methods of data compression. Each has limits in its ability to handle graphics information. The decision to use a RLE or Bounding Box compression scheme should be based on the likely distribution of bit-on data and the preferred ratio of data compression to access speed.

As mentioned earlier, in situations where memory is extremely tight, the Bounding Box method can profitably be used in conjunction with RLE or other algorithms. Of the various compression methods currently used in fast-access screen I/O environments, the Bounding Box method is recommended for maintaining maximum data compression while allowing the greatest access speed of bit-mapped font data.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 108.)

Vote for your favorite feature/article.
Circle Reader Service **No. 5.**

1989 DDJ INDEX

80386
Advanced memory management (N. Margulis), Apr., p. 24
Protected mode and multitasking under MS-DOS (T. Green), Sept., p. 64

A

Actor
For developing applications for MS Windows (J. Duntemann), Dec., p. 153
Writing filters in an object-oriented language (M. Franz), Dec., p.28
Multiple inheritance (M. Swaine), March, p. 107
Algorithms
Boyer-Moore and brute-force for string searches (C. Menico), July, p. 74
Autorouting with AI-based (A*) search algorithms (R. Nevin), Sept., p. 16
Simulated annealing and optimization (M.P. McLaughlin), Sept., p. 26
For precedence trees (M. Peterson), Sept., p. 44
LZW data compression algorithm (M. Nelson), Oct., p. 29
Sequential algorithm and multitasking (R. Kar), Nov., p. 16
ANSI C
From K&R to (S.R. Ladd), Aug., p. 74
Discussion of changes and uniformity (A. Stevens), Nov., p. 133
Update (A. Stevens) Dec., p. 144
Dennis Ritchie on (A. Stevens), C Sourcebook, p. 8
APL*PLUS/PC
Review of STSC's product (C. Burke), Feb., p. 72
Assemblers
MS-DOS reviewed (M. Schmidt), Jan., p. 70
Assembler code
Customizing for speed and reliability (M. Abrash, D. Illowsky), Sept., p. 52
Asynchronous communications
TINYCOMM (A. Stevens), Feb., p. 115
AWK
Writing AWK-like extensions to C (J. Mischel), June, p. 64
Revisited ("Letters"), Oct., p. 12

B

Back propagation
Neural networks and noise filtering (C. Klimasauskas), Jan., p. 32
Algorithm for multilevel neural net work (M. Swaine), Aug., p. 134
Benchmarks
an apologia (G. Vose, D. Weil), Feb., p. 36
C statements (D. Fox), Feb., p. 60
Rhealstone (R. Kar, K. Porter), Feb., p. 14
Turbo C and QuickC (S.R. Ladd), Aug., p. 89
Binary trees
Ordered by precedence trees (M. Peterson), Sept., p. 44
Vs. B-trees ("Letters"), Nov., p.8
Blackjack simulation
C programming (A. Stevens), July, p. 113
Boyer-Moore algorithm
For faster string searches (C. Menico), July, p. 74
Vs. brute force ("Letters"), Oct., p. 12
Breadth-first search algorithm
Autorouting with the A* algorithm (R. Nevin), Sept., p. 16
Brute-force method
Combined with Boyer-Moore for string searches (C. Menico), July, p. 74
Vs. Boyer-Moore ("Letters"), Oct., p. 12

C

C++
Compared to Modula-2 (S. Ladd), Jan., p. 62
Extended directory searches (J. Dlugosz), March, p. 55
Multitasking kernel (T. Green), Feb., p. 45
TAWK —a simple interpreter (B. Eckel), May, p. 50
The wave of tomorrow? (A. Stevens), Aug., p. 141
Implementing simulation systems (T. King), Sept., p. 40
Books, compilers, window object (A. Stevens), Sept., p. 121
Linked list class; some compilers and preprocessors (A. Stevens), Nov., p. 133
Home brew, table driven parser generator (J. Dlugosz), Dec., p. 40
Object-Oriented Program Design by M. Mullins (A. Stevens), Dec., p. 144
C to C++ (A. Stevens), C Sourcebook, p. 32
String Classes (S. Ladd), C Sourcebook, p. 8
C Interpreter
Build your own (H. Schildt), Aug., p. 38
SI: a C-like script interpreter (A. Stevens), May, p. 117
C interpreter for OS/2 and run-time linking (A. Schulman), Nov., p. 46
C Programming (Al Stevens)
Microsoft C bridge added to project, Jan., p. 111
TINYCOMM, Feb., p. 115
TINYCOMM begets SMALLCOM, March, p. 113
A phone directory and XModem for SMALLCOM, Apr., p. 109
SI: a C-like script interpreter, May, p. 117
Scripts for SMALLCOM, June, p. 119
Blackjack simulation program, July, p. 113
C++ the wave of tomorrow, and *typedef* misuse, Aug., p. 141
C++ books, compilers, a window object, and extensions, Sept., p. 121
C++ tools, menu classes, the string class, and ANSI, Oct., p. 123
Linked list class for C++, Nov., p. 133
Compilers and preprocessor, Nov., p. 133
TEXTSRCH (document retrieval system), ANSI update, Dec., p. 144
C Programming
"C"erious Toolkit, TSR Systems, review (K. Weiskamp), Apr., p. 92
Writing AWK-like extensions to C (J. Mischel), June, p. 64
And Smalltalk (D. Thomas, R. Best), August, p. 16
C-to-Fortran —C tools and Fortran libraries (M. Floyd), Aug., p. 22
Conversion medium (PCX images to Postscript printers) (K. Quirk), Aug., p. 30
Building your own C interpreter (H. Schildt), Aug., p. 38
C multidimensional arrays at run time (P. Anderson), Aug., p. 50
C dynamic memory use (R. Merilatt), Aug., p. 62
C procedure tables (T. Berens), Aug., p. 68
ANSI C from K&R (S.R. Ladd), Aug., p. 74
A generic Heapsort algorithm in C (K. Russell), Aug., p. 81
Turbo C and QuickC (Examining Room, S.R. Ladd), Aug., p. 89
C Windows Toolkit & PCX Programmer's Toolkit (Examining Room, T. Castle), Aug. p. 91
Concurrent C for real-time programming (N. Gehani, W. Roome), Nov., p. 38
Concurrent C for discrete event simulation (N. Gehani, W. Roome), C Special, p. 24
C Windows Toolkit
Examining Room review (T. Castle), Aug., p. 91
cdev
And Object C on the Mac (B. Waters), Mac Journal, p. 50
Cellular automata
A new way of simulation (J. Coppola, F. Marchese), Sept., p. 34
Code optimization
Optimization technology for writing faster code (K. Rowe), June, p. 56
Color QuickDraw
Adding color and using multiple monitors (C. Derossi), Mac Journal, p. 8
Combinatorial optimization
Solving with the technique of simulated annealing (M.P. McLaughlin), Sept., p. 26
CommonView
Examining Room review (N. Bergman), Oct., p. 74
Communication
PC-to-minicomputer communication (M. Servello), Oct., p. 18
Hamming-code decoding (B. White), Oct., p. 52
Communications protocols
Finite state machine for XModem (D. Smith), Oct., p. 45
Kermit meets Modula-2 (B. Anderson), May, p. 22
TINYCOMM (A. Stevens), Feb., p. 115
Compaction technique
Memory allocation compaction system (S. Peterson), Apr., p. 50
Compilers
Routines for use with; QuickPak Professional; review (B. Tonkin), Apr., p. 96
Also see "Examining Room"
Container objects
Turbo Pascal, packaged in library modules (A. Hejlsberg), Nov., p. 56
Coprocessors
Multitasking OS and graphics (Amiga) (C. McManis), July, p. 36

D

Data acquisition
A real-time system (M. and M. Bunnell), June, p. 36
Data communications
Finite state machine for XModem (D. Smith), Oct., p. 45
PC-to-minicomputer communication (M. Servello), Oct., p. 18
Hamming-code decoding (B. White), Oct., p. 52
High-speed file transfers (C. Menico), Oct., p. 38
Data compression
All-purpose technique for programming toolbox (LZW) (M. Nelson), Oct., p. 29
RLE reduces storage requirements (R. Zigon), Feb., p. 126
Another approach to RLE (P. Daley), May, p. 130
More on RLE ("Letters"), Oct., p. 12
Data-flow
Diagrams —using mapping to improve the quality of (G. Lazarev), Oct., p. 61
Multitasking and sequential algorithm (R. Kar), Nov., p. 16
Debugging
TSR Programs with Turbo Debugger (C. Menico), Feb., p. 67
Edward K. Ream's Sherlock debugger (Examining Room, A. Lane) Apr., p. 91
With Borland's Turbo debugger (B. Catchings, M. Van Name), July, p. 64
C programs, (B. Edgar), C Sourcebook, p. 56
Demand paging
Demand paged virtual memory (K. Dahlgren), Apr., p. 32
Advanced 80386 memory management (N. Margulis), Apr., p. 24
DESQview
Using the API to develop the parallel *dvmake* (M. Streich), Nov., p. 28
Device drivers
That support true global variables (J. Mischel), Oct., p. 70
Loading with an INIT on the Mac (J. Rosford), Mac Journal, p. 15
Mac Device Manager, template with Think C (B. Waters), Mac Journal, p. 36
Directories
Extended searches using C++ (J. Dlugosz), March, p. 55
DOS
Three utilties for with Modula-2 (K. Porter), Jan., p. 120
Undocumented DOS (R. James), June, p. 26
DOS Exec
More memory through swapping calling programs (K. Kokkonen), April, p. 14
Dynamic link libraries (DLLs)
Under MS Windows (M. Johnson, M. Solinski), March, p. 28
Dynamic linking
Linking while the program is running (A. Schulman), Nov., p. 46

1989 DDJ INDEX

Dynamic memory
Coping with C dynamic memory problems (R. Merilatt), Aug., p. 62

E

Editors
VEdit Plus (Examining Room, T.A. Elkins), Aug., p. 90
An icon editor (K. Weiskamp, L. Heiny), July, p. 24
Eiffel
Its development and use (B. Meyer), Dec., p. 48
Multiple inheritance (M. Swaine), March, p. 107
EGA
Matching colors (K. Porter), April, p. 116
Examining Room product reviews
Microsoft's MASM 5.1, Borland's TASM 1.0, and SLR's OPTASM 1.5 (M. Schmidt), Jan., p. 70
STSC's APL*PLUS II (C. Burke), Feb., p. 72
Advanced Programming Institute's XVT Toolkit (M. Johnson), March, p. 70
Crescent Software's QuickPak (B. Tonkin), Apr., p. 96
Quick C vs. Turbo C (S.R. Ladd), May, p. 68; Aug., p. 89
Abraxas Software's PCYACC (A. Lane), June, p. 76
Borland's Turbo Debugger (B. Catchings, M. Van Name), July, p. 64
Genus's C Tools and PCX Programmer's Toolkit (T. Castle), Aug., p. 91
CompuView's VEdit Plus (T.A. Elkins), Aug., p. 90
Watcom C7.0 (J. Dlugosz), Sept., p. 74
CommonView (N. Bergman), Oct., p. 74
Silicon Graphics (SGI) parallelization scheme (B. Bauer), Nov., p. 72
Crescent Software's PDQ for QuickBasic (B. Tonkin), Dec., p. 88
Extended memory
Using on the PC AT (P. Thomson), Jan., p. 106
Extensible hashing
Keyed random access method (S. Heller), Nov., p. 66
Extensible Virtual Toolkit (XVT)
API's product reviewed (M. Johnson), March, p. 70
External Commands (XCMDs)
Quick drawing with (J. Anderson), May, p. 63

F

File transfers
High speed with NetBIOS (C. Menico), Oct., p. 38
Filters
With an object-oriented language (M. Franz), Dec., p. 28
Finite state machines
For XModem, but general enough for other protocols (D. Smith), Oct., p. 45
Programming ("Letters"), Nov., p. 12
Force-based simulations
C++ and implementing simulation systems (T. King), Sept., p. 40
Forth
A timed event network scheduler in (G. Ilg, R. Brown), Feb., p. 52
An extension of with TENS, Feb., p. 52
Forth Column, The (M. Tracy)
Forth news and Mandlebrot plotting program, Feb., p. 136
Fortran
Connecting with C (M. Floyd), Aug., p. 22
Functional Programming
Report on FPCA '89 and Haskell language (R. Fischer), Dec., p. 96

G

Global variables
A global variable device driver (J. Mischel), Oct., p. 70
Graphics
Coprocessors with multitasking OS (Amiga) (C. McManis), July, p. 36
Line-of-best-fit (W. Murray, C. Pappas), July, p. 14
An Icon editor (K. Weiskamp, L. Heiny), July, p. 24
Image mathematics (V. Duvanenko), July, p. 45
Graphics Programming (K. Porter)
How to draw a pixel on the EGA/VGA, Feb., p. 121
Lines galore, March, p. 118
Matching colors on the EGA, Apr., p. 116
GRAFIX viewport, May, p. 124
A routine that senses pixels; filling closed figures, June, p. 124
Virtual coordinates, July, p. 119
And curves, Aug., p. 146
Greedy algorithm
For locating a global optimum in a discrete space (M.P. McLaughlin), Sept., p. 26

H

Hamming-code theory
In "Hamming-Code Decoding" (B. White), Oct., p. 52
Hardenbergh, Hal
Interview with about neural nets (M. Swaine), July, p. 100
Haskell language
In report on Functional Programming Conference '89 (R. Fischer), Dec., p. 96
Heapsort algorithm
A sorting tool for any language (K. Russell), Aug., p. 81
Hypercard
XCMDs provide access to QuickDraw (J. Anderson), May, p. 63

I

Icon editor
Roll your own icons w/mouse (K. Weiskamp, L. Heiny), July, p. 24
Image processing
Arithmetic functions to combine and enhance images (V. Duvanenko), July, p. 45
Index
DDJ 1988 (K. Ralston), March, p. 93
INIT
Loading device drivers with, on the Mac (J. Rosford), Mac Journal, p. 15
Interprocess communications
In OS/2 (R. Duncan), June, p. 15
Interpreters
Build your own (H. Schildt), Aug., p. 38
SI: a C-like script interpreter (A. Stevens), May, p. 117
C interpreter for OS/2 and run-time linking (A. Schulman), Nov., p. 46
Small languages for specific tasks (M. Abrash, D. Illowsky), Sept., p. 52

K

K&R
Differences with ANSI C (S.R. Ladd), Aug., p. 74
KRAM
Keyed random access method —extensible hashing (S. Heller), Nov., p. 66
Kermit
Meets Modula-2 (B. Anderson), May, p. 22

L

LabView
Visual OOP with (R. Dye), Mac Journal, p. 30
Languages
Variable-level programming (R. Fischer), June, p. 46
Small languages for specific tasks (M. Abrash, D. Illowsky), Sept., p. 52
Linked lists
A class for C++ (A. Stevens), Nov., p. 133
Logic gates
And neural networks (T. King), Jan., p. 15

M

Macintosh
XVT for developing portable applications (M. Johnson), March, p. 70
Color QuickDraw (C. DeRossi), Mac Journal, p. 8
Avoiding INIT collisions (J. Rosford), Mac Journal, p. 14
Memory management with MacAPP (C. Bianchi), Mac Journal, p. 22
Visual object-oriented programming (R. Dye), Mac Journal, p. 30
Writing Macintosh device drivers (B. Waters), Mac Journal, p. 36
Persistent objects (C. Rovira), Mac Journal, p. 40
Wizardcopy for fast backups (D. Gaspar), Mac Journal, p. 44
Object C and the Macintosh control panel (B. Waters), Mac Journal, p. 50
On being or becoming a Macintosh developer (J. Custer), Mac Journal, p. 54
XCMDs provide access to QuickDraw (J. Anderson), May, p. 63
Network graphs in Object Pascal (S. Kienle), Dec., p. 17
make
dvmake with the DESQview API (M. Streich), Nov., p. 28
MASM
Microsoft's assember (Version 5.1) reviewed (M. Schmidt), Jan., p. 70
Mathematics
Arithmetic functions can enhance imaging (V. Duvanenko), July, p. 45
Memory
Using extended memory on the PC AT (P. Thomson), Jan., p. 106
More memory for DOS *Exec* (K. Kokkonen), Apr., p. 14
Advanced 80386 memory management (N. Margulis), Apr., p. 24
Demand paged virtual memory (K. Dahlgren), Apr., p. 32
SWAP —application-independent method (N. Mak), Apr., p. 44
Memory allocation compaction system (S. Peterson), Apr., p. 50
Management with MacApp (C. Bianchi), Mac Journal, p. 22
Microsoft C
Surrogate library makes "C Programming" project compatible with MSC, (A. Stevens), Jan., p. 111
Mini-interpreters
Define small languages for specific tasks (M. Abrash, D. Illowsky), Sept., p. 52
Minilanguages
Roll your own with mini-interpreters (M. Abrash, D. Illowsky), Sept., p. 52
Modula-2
Compared to C++ (S.R. Ladd), Jan., p. 62
Three utilities for DOS (K. Porter), Jan., p. 120
Meets Kermit (B. Anderson), May, p. 22
MS-DOS
Assemblers compared (M. Schmit), Jan., p. 70
Real-time modeling with (D. Bowling), Feb., p. 26
Application-independent method (N. Mak), Apr. p. 44
Undocumented DOS (R. James), June, p. 26
MS Windows
DLLs under (M. Johnson, M. Solinski), March, p. 28
XVT for developing portable applications (M. Johnson), March, p. 70
Microsoft Windows SDK and Actor (J. Duntemann), Dec., p. 153
Multitasking
C ++ kernel (T. Green), Feb., p. 45
80386 protected mode (T. Green), Sept., p. 64

1989 DDJ INDEX

Multitasking OS
And graphics coprocessors (Amiga) (C. McManis), July, p. 36
Real-time data acquisition (M. and M. Bunnell), June, p. 36
Interprocess communications in OS/2 (R. Duncan), June, p. 14
Multiple inheritence
Is it necessary? (M. Swaine), March, p. 107
Writing correct software with Eiffel (B. Meyer), Dec., p. 48
Are the emperor's new clothes object oriented? (S. Guthery), Dec., 80

N

NetBIOS
For high-speed file transfers (C. Menico), Oct., p. 38
Network graphs
In Object Pascal (S. Kienle), Dec., p. 17
Network windowing
Using the X Window System (J. Gettys), March, p. 42
The OSF windowing system (K. Hinckley), March, p. 78
Networking protocols
Developing with Unix Streams (M. Garwood, A. Schweig), Jan., p. 50,
Neural networks
And noise filtering (C. Klimasauskas), Jan., p. 32
Artificial neural networks & signal processing (S. Melnikof), Jan., p. 36
For signal processing (S. Melnikof), Jan., p. 36
Using for pattern recognition (T. King), Jan., p. 14
Interview with H. Hardenbergh about (M. Swaine), July, p. 100
Background on backpropagation (M. Swaine), Aug., p. 134
History of (M. Swaine), Sept., p. 114
MINOS II and ADAM I, early neural nets (M. Swaine), Nov., p. 124
Noise filtering
And neural nets (C. Klimassauskas), Jan., p. 32

O

Object-oriented programming
Multiple inheritance (M. Swaine), March, p. 107
Basic concepts in OOP (M. Floyd), Apr., p. 58
With Turbo Pascal (M. Floyd), July, p. 56
Smalltalk, the legend (J. Duntemann), Aug., p. 154
Pascal objects (J. Duntemann), Sept., p. 128
Writing filters (M. Franz), Dec., p. 28
With Eiffel (B. Meyer), Dec., p. 48
Simulated bench environment—LogicLab (K. Ayers), Dec., p. 72
Are the emperor's new clothes object oriented? (S. Guthery), Dec., p. 80
OOPSLA '89 conference report (M. Swaine), Dec., p. 140
Visual OOP with LabView (R. Dye), Mac Journal, p. 30
Object C and Mac *cdevs* (B. Waters), Mac Journal, p. 50
Object Pascal
And network graphs (S. Kienle), Dec., p. 17
MS QuickPascal user interface (J. Mouhanna, M. Vose), Dec., p. 64
OOPSLA '89
Report on conference (M. Swaine), Dec., p. 140
Open Software Foundation (OSF)
Windowing system (K. Hinckley), March, p. 78
OPTASM
SLR's assembler (Version 1.5) reviewed, Jan., p. 70
Optical character recognition system (OCR)
And neural networks (T. King), Jan., p. 14
Optimization
Writing faster code (K. Rowe), June, p. 56
In a parallel environment (Examining Room., B. Bauer), Nov., p. 72
OS/2
Interprocess communications facilities (R. Duncan), June, p. 15
Run-time dynamic linking, mini C interpreter (A. Schulman), Nov., p. 46

P

Pascal
With objects (J. Duntemann), Sept., p. 128
Parsers
Generating with PCYACC, review (A. Lane), June, p. 76
Table-driven generator (J. Dlugosz), Dec., p. 40
Pattern recognition
Using neural networks for (T. King), Jan., p. 14
PC AT
Using extended memory on (P. Thomson), Jan., p. 106
PCX files
Translating to Postscript printers with C (K. Quirk), Aug., p. 30
PCX Programmer's Toolkit
Examining Room review (T. Castle), Aug., p. 92
Persistent objects
For writing database management systems in Smalltalk (C. Rovira), Mac Journal, p. 40
PEX
3-D graphics for X, March, p. 49
Pixels
How to draw on EGA/VGA (K. Porter), Feb., p. 121
A routine that senses pixels (K. Porter), June, p. 124
Polymorphism
Using it with OOP in Pascal (J. Duntemann), Nov., p. 142
Precedence trees
Improved search strategies for higher performance (M. Peterson), Sept., p. 44
Presentation Manager (PM)
Application template (H. Schildt), March, p. 16
Putting mouse and graphics to work (B. Murray, C. Pappas), July, p. 14
OSF Motif (K. Hinckley), March, p. 78
Programming Paradigms (M. Swaine)
Future programming environments, Jan., p. 124
Inheritence dilemma, Feb., p. 108
Multiple inheritance, March, p. 107
Lisp; superlinearity speedup; teaching of programming, Apr., p. 98
Interview with Robert Floyd, May, p. 107
The mainstreaming of OOP, June, p. 114
Interview with Hal Hardenbergh—neural nets, July, p. 100
Background on back propagation, Aug., p. 134
More on neural nets, history of, Sept., p. 114
Interview with Dave Parker on backprop, Oct., p. 112
MINOS II and ADAM I, early neural nets, Nov., p. 124
Report on OOPSLA '89, Dec., p. 140
Prolog
Executable specifications with (G. Lazarev), Oct., p. 61
Protected mode
On the 80386 for memory protection and multitasking (T. Green), Sept., p. 64
Puzzles
The Puzzling Adventures of Dr. Ecco—book review (J. Amsterdam), Apr., p. 96

Q

QuickBasic
Code runs fast with PDQ (Examining Room., B. Tonkin), Dec., p. 88
QuickC
Vs. Turbo C—review (S.R. Ladd), May, p. 68
Vs. Turbo C benchmark sum up—review (S.R. Ladd), Aug., p. 89
QuickPascal
MS QuickPascal user interface (J. Mouhanna, M. Vose), Dec., p. 64

R

Real time
Development of kernels with TENS (G. Ilg, R.J. Brown), Feb., p. 52
Modeling with MS-DOS (D. Bowling), Feb., p. 26
Rhealstone benchmark (R. Kar, K. Porter), Feb., p. 14
Real-time data acquisition (M. and M. Bunnell), June, p. 36
tty controller in Concurrent C (N. Gehani, W. Roome), Nov., p. 38
Rhealstone
A real-time benchmarking proposal (R. Kar, K. Porter), Feb., p. 14
Run length encoding (RLE)
Reduces storage requirements (R. Zigon), Feb., p. 126
Another approach (P. Daley), May, p. 130
More on ("Letters"), Oct., p. 12
Run-time
Dynamic linking under OS/2, with a mini C interpreter (A. Schulman), Nov., p. 46

S

Sbench
Program generator for better C code (Fox), Feb., p. 60
Sector copying utility
WizardCopy for the Mac (D. Gaspar), Mac Journal, p. 44
Simulated annealing
Solving combinatorial optimization (M. McLaughlin), Sept., p. 26
Cellular automata (J. Coppola, F. Marchese), Sept., p. 34
Simulated bench environment
LogicLab system (K. Ayers), Dec., p. 72
Simulations
Force-based simulations and C++ (T. King), Sept., p. 40
Signal processing
And artificial neural networks (S. Melnikof), Jan., p. 36
SMALLCOM
Extension of TINYCOMM (A. Stevens), March, p. 113
With file transfer protocols and XModem (A. Stevens), April, p. 109
With a script interpreter (A. Stevens), May, p. 115
Scripts for (A. Stevens), June, p. 119
Smalltalk
The ultimate object-oriented language (J. Duntemann), Aug., p. 154
+ C to add power to your programs (D. Thomas, R. Best), Aug., p. 16
OO LogicLab system, simulated bench environment (K. Ayers), Dec., p. 72
For PM (J. Duntemann), Dec., p. 153
Persistent objects for database management (C. Rovira), Mac Journal, p. 40
Software development
For the Mac (J. Custer), Mac Journal, p. 54
Software metrics
Coping with complex programs (K. Siyan), March, p. 60
String searches
Writing AWK-like extensions to C (J. Mischel), June, p. 64
Boyer-Moore algorithm combined with brute-force (C. Menico), July, p. 74
Structures
Language-independent dynamic pseudostructures (B. Tonkin), May, p. 39
Structured Programming (J. Duntemann)
Modula-2 put to work with three utlities for DOS (K. Porter), Jan., p. 120
Return of the shower curtain salesman, Feb., p. 130

Text screen metrics and Comdex, March, p. 123
Speed windows graphics and virtual screens, Apr., p. 120
Type casting in Turbo Pascal and Modula-2, May, p. 132
Writing assembly code with HLLs, June, p. 130
OOP —the next generation of structured methods, July, p. 128
Smalltalk the ultimate object-oriented language, Aug., p. 154
Pascal with objects, Sept., p. 128
Object-oriented glossary, Oct., p. 132
Polymorphism and OOP in Pascal, Nov., p. 142
MS Windows, Actor, Smalltalk/V PM and OOP, Dec., p. 153

SWAP
Application-independent method for MS-DOS (N. Mak), Apr., p. 44

System security
Avoiding file system vulnerability and threats to network security (D. Moir), June, p. 75

T

TASM
Borland assembler (Version 1.0) reviewed, Jan., p. 70

TAWK
A simple interpreter in C++ (B. Eckel), May, p. 50

Text screen metrics
Text-oriented programs need to be video adapter "aware" (J. Duntemann), March, p. 123

Timed event network scheduler (TENS)
In Forth (G. Ilg, R. Brown), Feb., p. 52

TINYCOMM
The tiny communications program (A. Stevens), Feb., p. 115

TSR (Terminate-and-Stay-Resident) programs
Debugging, Feb., p. 67
Part 1, creating TSRs with Turbo Pascal (K. Pottebaum), May, p. 14
Part 2, putting common TSR tasks to work (K. Pottebaum), June, p. 72

Turbo C
To MSC surrogate library (A. Stevens), Jan., p. 111
Vs. QuickC —review (S. Ladd), May, p. 68
Vs. QuickC benchmark sum up —review (S. Ladd), Aug., p. 89

Turbo Debugger
Using Borland's product to debug TSRs (C. Menico), Feb., p. 67
Examining Room (B. Catchings, M. Van Name), July, p. 64

Turbo Pascal
More memory for DOS *Exec* (K. Kokkonen), April, p. 14
Part 1, creating TSR programs with (K. Pottebaum), May, p. 14
Part 2, putting common TSR tasks to work (K. Pottebaum), June, p. 72
Turbo Pascal with Objects (M. Floyd), July, p. 56

U

Unix
OSF windowing system (K. Hinckley), March., p. 78
Streams (M. Garwood, A. Schweig), Jan., p. 50

V

Variable-level programming
Languages that accomodate a range of needs (R. Fischer), June, p. 46

VEdit Plus
Examining Room review (T.A. Elkins), Aug., p. 90

Virtual coordinates
In graphics programming (K. Porter), July, p. 119

W

Watcom C7.0
Review (J. Dlugosz), Sept., p. 74

Windows
QuickPascal's OO user interface (J. Mouhanna, M. Vose), Dec., p. 64
Microsoft Windows SDK and Actor (J. Duntemann), Dec., p. 153

WizardCopy
Sector copying utility for the Mac (D. Gaspar), Mac Journal, p. 44

X

X/GEM
Writing portable applications with (B. Fitler), March, p. 38

XCMDs
External commands —quick drawing with (J. Anderson), May, p. 63

XModem
A finite state machine for (D. Smith), Oct., p. 45

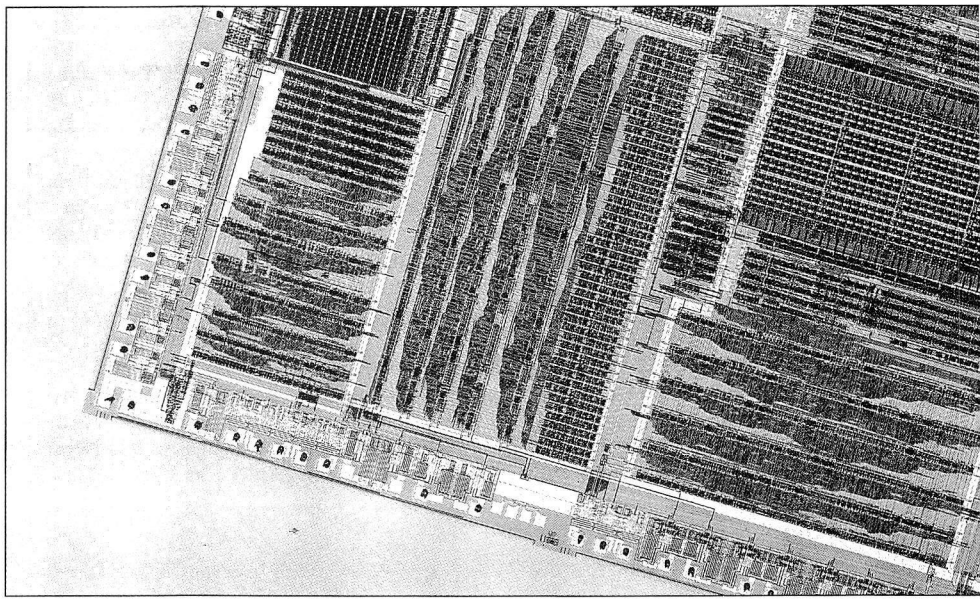
XVT
See Extensible Virtual Toolkit

X Window System
Network windowing using (J. Gettys), March, p. 42
PEX 3-D graphics for, March, p. 49

VESA VGA BIOS Extensions

A software standard for Super VGA

Bo Ericsson



An integral part of IBM's PS/2 announcement in April 1987 was the video graphics array (VGA) system. Based on the architecture of the enhanced graphics adapter (EGA), the VGA offered extended resolutions and a new 256-color video mode. Since that time, the VGA has grown in importance and is today an established PC video standard. As a matter of fact, all "old" video standards — the monochrome display adapter (MDA), color graphics adapter (CGA), Hercules Graphics Adapter, and EGA — are quickly losing ground to the VGA.

There are several reasons for the VGA's success. For one thing, the new VGA resolutions (see Figure 1), together with lower-priced multi-frequency monitors, have made the VGA a more attractive solution than previous standards. Also, a multitude of VGA offerings and fierce competition have made a baseline VGA an economically attractive choice.

As a matter of fact, competition in the VGA marketplace not only has driven the prices of VGA boards to the bottom, but has pushed up the features and capabilities of these boards. Virtually all VGA controllers available today are compatible down to the register level with the IBM VGA, and almost all of them implement some extensions to the IBM VGA.

The term "Super VGA" is used in this article to identify video hardware that implements a full superset of the standard VGA, including register compatibility. Extensions to the IBM VGA can be classified into three different categories:

1. Backwards compatibility
2. Functional extensions
3. Higher spatial and color resolutions

Backwards Compatibility

The basic IBM VGA is, at best, compatible with older video standards only at the BIOS level. There is a large population of older programs written specifically for, and directly to, the CGA or Hercules Graphics Adapter that bypass the BIOS partially or completely. Because of this, none of these applications run on a standard VGA.

However, most VGA products offer some register-level support for these older standards. These implementations either attempt to automatically detect older programs and switch into a suitable compatible video mode or require a utility program to lock the video hardware into a compatible video mode.

Functional Extensions

The basic VGA is a pretty dumb device; the CPU (that is, the application program) is required to do almost all graphics processing. Only certain logical operations on the graphics data can be performed by the standard VGA hardware. There are no functions for *BitBlts* (bit-block-transfers), line drawing, and so on.

Bo is a software engineering manager at Chips and Technologies Inc., 3050 Zanker Road, San Jose, CA 95134.

In graphics-intensive applications, such as MS-Windows, OS/2 Presentation Manager, and GEM, manipulating the graphics bitmap takes considerable time and affects system performance. For this reason, several VGA controller vendors have put various graphics capabilities directly into the VGA hardware.

For instance, certain VGA controllers implement a graphics cursor in hardware. All graphics user interfaces (such as Windows, GEM, X-Windows, Presentation Manager, etc.) use a graphics cursor. The graphics cursor is an icon (usually an arrow) that moves around the screen as the mouse is moved. A lot of CPU processing is required to move the graphics cursor even one pixel on the screen. Instead of refreshing the actual bitmap on a standard VGA, these controllers need only the coordinate of the "hot-spot." The actual display of the cursor is done in hardware; bitmap manipulation is not necessary.

Other VGA controllers implement more sophisticated write modes, elementary *BitBlt* capabilities, or other functions that relieve the CPU of some graphics processing.

Higher Resolutions, More Colors

The most exciting aspect of all Super VGA implementations, however, is the higher resolutions and the increased number of simultaneous colors on the screen. The standard VGA can display 16 simultaneous colors in 640 × 480 resolution and 256 colors in 320 × 200, as described in Figure 1. In contrast, a typical Super VGA board can do 1024 × 768 in 16 colors and 640 × 480 in 256 colors. In the near future, a range of VGA controllers will be able to do 1024 × 768 in 256 colors. And a little further down the line, some controllers will have the capability of 1280 × 1024 resolution in 16 colors.

Developments in the monitor market make these extended resolutions especially important. Multifrequency monitors capable of resolutions up to 1024 × 768 are available today for less than $1000, and the price is expected to drop even further.

Planar vs. Packed Pixel Modes

Before beginning a discussion on Super VGA graphics, a brief summary of the basic video memory modes is required. VGA graphics video modes use either planar or packed pixel video memory architecture.

In planar mode, the video memory is divided into four separate planes. One pixel is defined by 4 bits, 1 bit per plane. Eight pixels are defined by 4 bytes, 1 byte per plane. Because one pixel is defined by 4 bits, 16 colors can simultaneously be displayed.

Normally, only one plane can be accessed at one time by the CPU. To access another plane, the hardware registers of the VGA have to be reprogrammed. For rapid fills of a large area to a certain color, the VGA can be programmed for 32-bit operation, allowing simultaneous access to all four planes.

In packed pixel mode, only one memory plane is available. One pixel is defined by 1 byte in the memory, yielding 256 simultaneous colors.

The Developer's Dilemma

In spite of this revolution and the fantastic opportunities that Super VGA provides, software development has been slow in tapping into the capabilities. Very few applications have Super VGA support, and only OEM-specific display drivers (software tied directly to a certain VGA controller) can generally exploit Super VGA resolutions and capabilities.

There are several reasons why software development for Super VGA has been sluggish. The most important reason is that almost all Super VGA hardware implementations are different from one another — a Super VGA controller from manufacturer A is usually significantly different from manufacturer B's because no common hardware or software interface exists.

The software developer has to gather a significant understanding of intimate details of each Super VGA controller (of which there are at least ten at present) and each implementation (of which there are dozens, maybe hundreds) that he/she intends to support. The cost of acquiring this knowledge and supporting these disparate environments is prohibitively high; software developers have shunned Super VGA for this reason.

Non-standard Initialization

Super VGA implementations differ significantly in the video mode initialization procedure. One piece of mode setting code will not work on more than one Super VGA board because the I/O addresses for the extended registers required for Super VGA operation vary from implementation to implementation. In addition, the specific parameters for the registers all depend on the VGA controller.

Another aspect of this problem is that there is no uniform BIOS support for mode initialization across Super VGA products. No video mode number scheme exists. A 640 × 480 256 color video mode is called 79 in one implementation and 43 in another. Also, no standardized mode initialization call exists.

All this means that an application cannot program the hardware directly (because no standard hardware exists), nor can it call a BIOS to initialize the mode (because a standardized mode number doesn't exist, and because no standardized calling sequence is established).

Different Windowing Schemes

Another area where Super VGA implementations differ greatly is in how the video memory is accessed. In the IBM PC, a maximum of 128K is devoted to the video system. This address space is located between A0000 and BFFFF hex. For compatibility reasons, only the 64K at A0000 is normally used for

	Resolutions					
Colors	320 × 200	640 × 200	640 × 350	640 × 480	800 × 600	1024 × 768
2	CGA	CGA	EGA	VGA	Super VGA	Super VGA
4	CGA	EGA	EGA	VGA	Super VGA	Super VGA
16	EGA	EGA	EGA	VGA	Super VGA	Super VGA
256	VGA	Super VGA	Super VGA	Super VGA	Super VGA	Super VGA

Figure 1: *PC graphics resolutions and colors*

Resolution	Colors	Pixels	Bits per pixel	Total memory (bytes)	Planes	CPU memory (bytes)
640 × 480	16	307200	4	153600	4	38400
800 × 600	16	480000	4	240000	4	60000
1024 × 768	16	786432	4	393216	4	98304
640 × 400	256	256000	8	256000	1	256000
640 × 480	256	307200	8	307200	1	307200
800 × 600	256	480000	8	480000	1	480000
1024 × 768	256	786432	8	786432	1	786432

Figure 2: *Memory requirements of Super VGA modes*

Super VGA resolutions (another video board in the system might be located at B0000-BFFFF).

However, Super VGA video modes consume more video memory than is available in the CPU address space. Figure 2 details typical memory requirements of Super VGA modes. As is evident from this table, there has to be a mechanism for the CPU to reach into the video memory using the 64K (or 128K) "window" available in the CPU address space.

Unfortunately, there are almost as many windowing schemes as there are Super VGA controllers. Some controllers have one window into the video memory, while others have two. Some controllers have separate read and write windows, while others allow read/write in both windows. Some controllers implement a "sliding" windowing scheme, whereby a window can be placed on any boundary in the video memory, while others allow placement of the window only on a 64K boundary.

On top of this, the hardware registers that control the windowing scheme are located at different I/O addresses and require different parameters.

Enter the VESA BIOS Extension

The Super VGA BIOS extension standard, as defined by the Video Electronics Standards Association (VESA), intends to remedy the incompatibility issues addressed earlier. The standard tries to address all major problems a software developer faces when writing software for Super VGA.

Technically, the VESA BIOS extension is implemented as an addition to the regular video BIOS, accessed through software interrupt 10 hex. Standard video BIOS functions are called by placing function numbers in the range from 0 to 1C hex, depending on the function, in the AH CPU register and then generating a software interrupt 10 hex. To call a VESA BIOS function, the application would place the value 4F hex in the AH register, place a function number in the AL register, and then generate an interrupt 10 hex. Figure 3 describes the VESA BIOS extension functions.

The VESA BIOS extension may be placed in ROM together with the regular BIOS. It may also be implemented as a device driver, loaded by the operating system at boot time. Initially, most VESA BIOS extensions will be available as TSR programs. To the application, the method of implementation is irrelevant; functionally, the BIOS extension behaves the same.

The VESA BIOS extension provides two fundamental services to the application program:

1. Information
2. Hardware setup

Global Information

To be able to adapt to a specific Super VGA environment, an application needs several important pieces of information. First and foremost, an application needs to know whether the specific environment is indeed capable of Super VGA resolutions. The application also needs to know whether any VESA support is available. In addition, certain applications might want to identify a specific VGA controller.

This kind of global information is provided by VESA BIOS function 0, *Return Super VGA mode information*. Before the application calls this function, it has to allocate a buffer of 256 bytes. The VESA BIOS extension will fill this buffer with various types of information.

One of the most important pieces of information returned by function *0* is a pointer to a list of Super VGA modes supported by the display adapter. These video modes can be VESA-defined modes as well as OEM-defined modes. See Figure 4 for a list of VESA-defined video modes.

Mode-specific Information

To determine the characteristics of a particular video mode, the application would then call VESA BIOS function 1, *Return Super VGA mode information*. Like function *0*, the application has to allocate a 256-byte buffer prior to making the function call.

On return from the function, the VESA BIOS extension will have filled a struc-

The following functions are defined by the VESA BIOS extension. They are all accessible through interrupt 10 hex with AH set to 4F hex.

Every function returns status information in the AX register. The format of the status word is as follows:

```
AL== 4Fh:      Function is supported
AL!= 4Fh:      Function is not supported
AH== 00h:      Function call successful
AH== 01h:      Functiion call failed
```

Function 0 - Return Super VGA Information

```
Input:    AH=4Fh    Super VGA support
          AL=00h    Return Super VGA information
          ES:DI=    Pointer to information block
Output:   AX=       Status
          All other registers are preserved
```

The information block has the following structure:

```
VgaInfoBlock struc
   VESASignature   db   'VESA'      ; 4 signature bytes
   VESAVersion     dw   ?           ; VESA version number
   OEMStringPtr    dd   ?           : Pointer to OEM string
   Capabilities    db   4 dup(?)    ; capabilities of the video environment
   VideoModePtr    dd   ?           ; pointer to supported Super VGA modes
VgaInfoBlock ends
```

Function 1 - Return Super VGA mode information

```
Input:    AH=4Fh    Super VGA support
          AL=01h    Return Super VGA information
          CX=       Super VGA video mode
          ES:DI=    Pointer to information block
Output:   AX=       Status
          All other registers are preserved
```

Function 2 - Set Super VGA video mode

```
Input:    AH=4Fh    Super VGA support
          AL=02h    Set Super VGA video mode
          BX=       D0-D14 = video mode
                    D15 = Clear memory flag
                          0 = Clear video memory
                          1 = Don't clear video memory
Output:   AX=       Status
          All other registers are preserved
```

Function 3 - Return current video mode

```
Input:    AH=4Fh    Super VGA support
          AL=03h    Return current video mode
Output:   AX=       Status
          BX=       Current video mode
          All other registers are preserved
```

Figure 3: *VESA BIOS extension functions (accessible through interrupt 10 hex with AH set to 4F hex)*

ture, called the *ModeInfoBlock*, with all relevant information about this video mode. See Figure 5 for a description of the *ModeInfoBlock*.

Mode Attributes

The first word (16 bits) in the *ModeInfoBlock*, the *ModeAttributes* field, specifies several important characteristics of the video mode. See Figure 6 for the layout of this field.

Bit *D0* in the *ModeAttributes* field specifies whether the mode is supported by the present hardware configuration. If a particular video mode requires a certain monitor, and this monitor is presently not connected to the system, this bit can be cleared to block access to the mode. Applications should never try to initialize a video mode whose *ModeAttributes D0* is set to *0*.

As will be evident in the discussion later, the VESA BIOS function *0* returns a lot of information to the application. Some of this information is mandatory, some is optional. Bit *D1* of the *ModeAttributes* specifies whether any optional information is available.

Bit *D2* indicates whether the output functions (TTY output, set/get pixel, scroll window, etc.) of the regular video BIOS can be used in this video mode. It is not mandatory for a VESA BIOS extension to support all or any output functions in Super VGA modes. The primary reason for this is that high-performance applications handle all output themselves anyway, for performance reasons. The fact that output support consumes a lot of precious memory space in a ROM-based implementation was also important in making this support optional. If bit *D2* is cleared, then no output support is available.

Bit *D3* specifies whether the mode is monochrome (*D3=0*) or color (*D3=1*). Bit *D4* defines the mode as either text mode (*D4=0*) or graphics mode (*D4=1*).

Window Description

The characteristics of the windowing system are described in the next field in the *ModeInfoBlock* structure. The *WinAAttributes* and *WinBAttributes* identify whether window A and B exist and are readable or writeable. All Super VGA boards capable of resolutions beyond 640 × 400 in 256 colors and 800 × 600 in 16 colors have at least one window into the video memory. Applications can determine the existence of a second window by testing bit *D0* of *WinBAttributes*.

The *WinGranularity* identifies the smallest address boundary that the window can be placed upon. In today's Super VGA boards, this varies from 1K to 64K. The *WinSize* field identifies the size of the windows. In a single-window system, the size is normally 64K, while in a dual window system, the size is normally 32K.

The location of the windows within the CPU address space is specified by the fields *WinASegment* and *WinBSegment*. Normally Window A is located at address A0000. If a second window is present, it would typically be located at A8000 or B0000. If the VGA controller implements different read and write windows, the second window could be located at the same CPU address as the first window. In such a system, a CPU read will access the read window, while a CPU write will access the write window.

The *WinFuncAddr* field specifies a direct address to the windowing function (Figure 3, VESA BIOS function 5). The standard way to access the video BIOS and the VESA BIOS extension is to generate an *int 10*. However, due to the large number of subfunctions using *int 10*, function dispatching may take considerable time. This makes *int 10* too slow for some graphics operations. One such time-critical operation is changing the windowing registers. By using the absolute address to the function, an application can issue a *far*

Function 4 - Save/Restore Super VGA video state

Input:	AH=4Fh	Super VGA support
	AL=04h	Save/restore Super VGA video state
	DL=00h	Return save/restore state buffer size
	CX=	Requested states
		D0=Save/restore video hardware state
		D1=Save/restore video BIOS data state
		D2=Save/restore video DAC state
		D3=Save/restore Super VGA state
Output:	AX=	Status
	BX=	Number of 64-byte blocks to hold the state buffer
	All other registers are preserved	
Input:	AH=4Fh	Super VGA support
	AL=04h	Save/Restore Super VGA state
	DL=01h	Save Super VGA video state
	CX=	Requested states (see above)
	ES:BX=	Pointer to buffer
Output:	AX=	Status
	All other registers are preserved	
Input:	AH=4Fh	Super VGA support
	AL=04h	Save/Restore Super VGA state
	DL=02h	Restore Super VGA video state
	CX=	Requested states (see above)
	ES:BX=	Pointer to buffer
Output:	AX=	Status
	All other registers are preserved	

Section 5 - CPU Video Memory Window Control

Input:	AH=4Fh	Super VGA support
	AL=05h	Super VGA video memory window control
	BH=00h	Select Super VGA video memory window
	BL=	Window number
		0=Window A
		1=Window B
	DX=	Window position in video meory
Output:	AX=	Status
Input:	AH=4Fh	Super VGA support
	AL=05h	Super VGA video memory window control
	BH=01h	Return Super VGA video memory window
	BL=	Window number
		0=Window
		1=Window
Output:	AX=	Status
	DX=	Window position in video memory

Mode number	Resolution	Colors
100h	640 × 400	256
101h	640 × 480	256
102h	800 × 600	16
103h	800 × 600	256
104h	1024 × 768	16
105h	1024 × 768	256
106h	1280 × 1024	16
100h	1280 × 1024	256

Figure 4: *VESA-defined Super VGA modes*

call directly into it, speeding up execution considerably.

Optional Information

Only a portion of the *ModeInfoBlock* is obligatory information. The other section is optional and is provided if the specific mode is nonstandard. None of the modes defined by VESA (see Figure 4) require the optional information. For an OEM-specific mode, however, the VESA BIOS extension needs to inform the application about items such as screen resolution, number of planes, and bits per pixel.

Refer to the VESA BIOS extension specification for information on how to use these optional fields.

Video Mode Initialization

One main objective of the VESA BIOS extension is to help applications set up video modes. This is realized through VESA BIOS Function 2, *Set Super VGA video mode.* The application simply places the video mode to be initialized in the BX register and calls this function. Normally, the video memory will be cleared, but if the application sets bit *D15* of the BX register prior to calling the function, the memory will be preserved.

VESA mode numbers are 15 bits wide (see Figure 4). OEM-defined mode numbers are 7-bits wide and are implemented as a subset of VESA-defined modes. Due to this numbering convention, VESA modes, OEM-specific modes, and regular VGA modes can be initialized by using VESA BIOS Function 2.

If an application needs to know the present video mode, it would call VESA BIOS Function 3, *Return current video mode.* For applications (especially TSR programs) that need to interrupt other programs, the VESA BIOS Function 4, *Save/Restore Super VGA video state,* comes in handy.

The Windowing Function

Finally, the VESA BIOS extension provides a mechanism to control the position of the video memory windows. This is handled by Function 5, *CPU video memory window control.* To reposition a window into the video memory, the application simply places the window position in the DX register, the window number (*0* for Window A and *1* for Window B) in the BL register, and calls Function 5.

The window position is not specified as a byte offset, but rather in terms of granularity units. As stated earlier, the window granularity expresses the smallest boundary on which the window can be placed. Today's Super VGA boards have granularities between 4K and 64K. Thus, if the granularity is 16K, and the application wants to position the window at 64K, the window position is 64/16 = 4 granularity units.

Conclusion

The VESA BIOS extension provides all necessary information and programming support to Super VGA applications. For the first time, it is possible to develop generic graphics software, tapping into the exciting capabilities of Super VGA.

However, just because the VESA BIOS extension has made it possible to write such applications doesn't mean it will be trivial. Most of the complexity in dealing with Super VGA stems from managing windows into the video memory. Anyone already familiar with writing software for one Super VGA board should have no difficulty in programming others using the VESA BIOS extension.

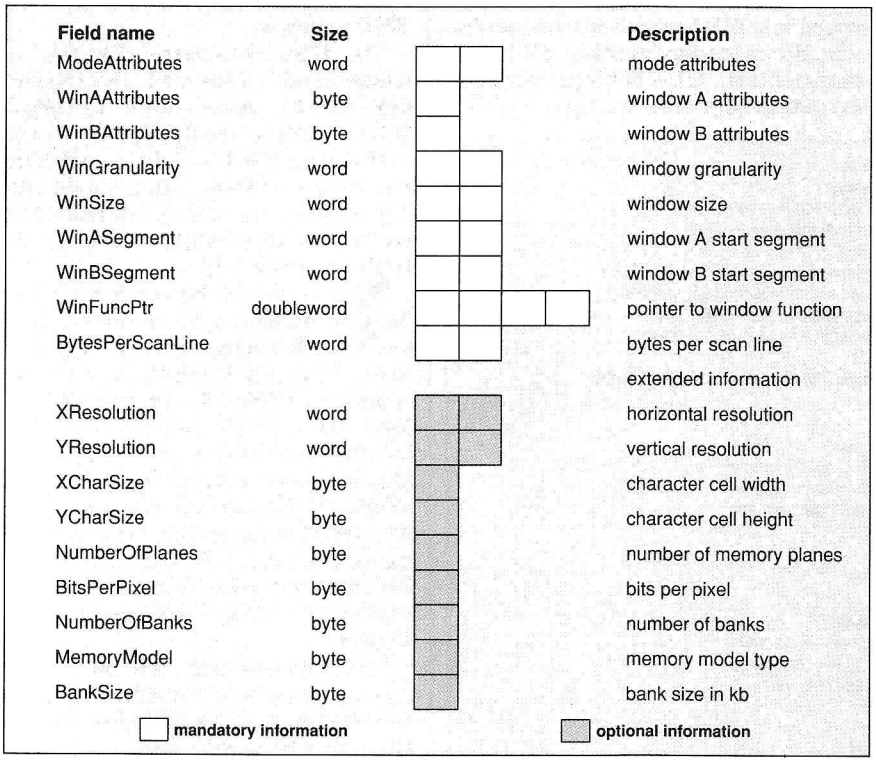


Figure 5: *Mode information block structure*

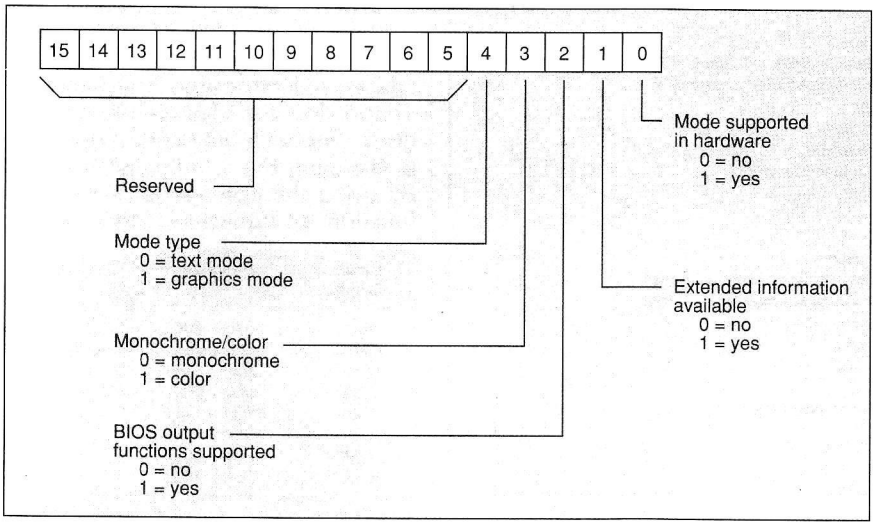
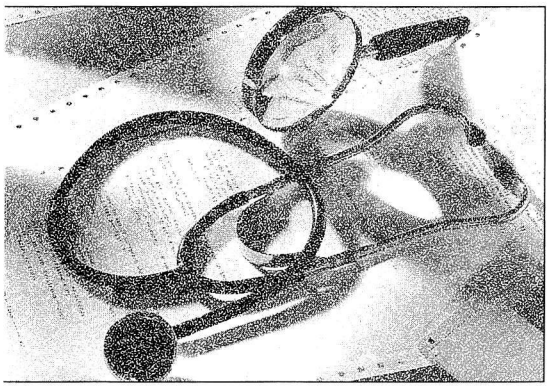


Figure 6: *Mode attribute field*

For More Information

Video Electronics Standard
1330 S. Bascom Ave., Ste. D
San Jose, CA 95128-4502
408-971-7525

DDJ
Vote for your favorite feature/article.
Circle Reader Service **No. 12.**



Cruising with TopSpeed

A full-featured toolset is the real value in this C compiler

Alex Lane

It is fashionable in some circles to yawn upon hearing that a new C compiler has hit the market. Such folks see the C world as divided into two main camps — the Microsofts and the Turbos — with a smattering of fanatics representing an insignificant fringe. That fringe, however, has lately succeeded in whipping up the C market, the result being a lively free-for-all as new arrivals such as Watcom, Zortech, and now TopSpeed C attempt to prove their worth to programmers. Readers, familiar with the TopSpeed name, will associate it with a popular Modula-2 compiler and a recently introduced Pascal compiler, marketed by Jensen & Partners International. JPI, a company established in 1987 by a group of former Borland employees, is a relatively small company with unquestionably large vision. They intend to develop an integrated multilanguage environment that will let programmers seamlessly mix and match routines from a broad spectrum of languages, including ISO Pascal, C, C++, Modula-2, and Ada. What JPI in effect proposes to do is to link each language dynamically into the system as an overlay at run time, thus allowing each language compiler to use the same optimizing code generator. If TopSpeed C is any indicator, JPI has its sights set on a worthwhile goal.

Alex is a knowledge engineer for Technology Applications Inc. in Jacksonville, Florida. He can be reached on BIX as a.lane or through MCI mail as ALANE.

What You Get

I reviewed the TopSpeed C, Version 1.02, Extended Edition, which is basically the standard package (comprised of an optimizing 100 percent ANSI C compiler and high-speed linker, an automatic *make* facility, an editing environment, and source-level debugger) combined with the TopSpeed C TechKit, which provides enhanced functionality in the form of library source code, Windows support, DOS dynamic linking, profiling, and post-mortem debugging, among other features. The standard TopSpeed package consists of seven diskettes and nearly three inches of paperback documentation, consisting of a user manual, a language reference, a library reference, and a language tutorial. The TechKit comes on an additional four disks and has separate documentation.

Finding Files

I dread installing software that needs to use DOS environment variables to find files on my disk. For one thing, I only have so much DOS environment space; for another, I often find that two different packages use the same DOS environment variable name in different ways, and that no amount of fiddling with SET statements shall allow the twain to meet on a consistent basis. If you have two or more C compilers installed on your hard disk, you probably know what I mean.

If you want TopSpeed C to use DOS environment variables, you can so specify by using the */y* flag from the command line, but why bother? I took an immediate liking to TopSpeed's redirection file feature, which acts as a sort of private environment. A sample redirection file, TS.RED, is reproduced in Figure 1. The syntax is similar to that of DOS paths. The first line indicates that all KABOOOOM.* files are found in the directory C:\ KABOOOOM. Analogously, all other *.C files may be found either in the current directory (denoted by the '.'), or in C:\TS \EXAMPLES, C:\TS\ SRC, or in D:\ FRAC \PROGS. The remaining lines are self-explanatory, except perhaps for the line that refers to *.A files that are TopSpeed assembler files.

By editing TS.REG, then automati-

```
KABOOOOM.* = C:\KABOOOOM
*.C       = .;    C:\TS\EXAMPLES; C:\TS\SRC; D:\FRAC\PROGS;
*.PRJ     = .;    C:\TS\PRJ; C:\TS\EXAMPLES;
*.H       = .;    C:\TS\INCLUDE;C:\TS\EXAMPLES; D:\FRAC\PROGS;
*.A       = .;    C:\TS\LIB; C:\TS\EXAMPLES; C:\TS\SRC

*.OBJ     =       C:\TS\OBJ; C:\TS\LIB;
*.LIB     = .;    C:\TS\LIB; D:\FRAC\PROGS;
*.DOC     = .;    C:\TS\DOC;
TS.RED    = .;    C:\TS\SYS;
```

Figure 1: *A sample redirection file*

cally saving and reloading it, you can change the file search (and storage) behavior of the environment on-the-fly. If, despite your best intentions, you repeatedly end up with all your *.c, *.obj, *.h, and *.exe files in one giant directory, this feature is for you.

Integrated Development Environment

The cornerstone of the integrated environment is, of course, the editor. Out of the box, TopSpeed's editor is configured to use the WordStar command set. You can change part or all of that by editing a configuration file.

Actually, the TopSpeed configuration file TSCFG.TXT affords the user quite a bit of control over not just the editor but the TopSpeed environment in general. The file is a 33K ASCII text file that defines the menu structure, every menu option, the editor commands, and even compilation error messages used by the TopSpeed system. This is the file you edit if you want to make the environment editor act more like, say, Brief than WordStar. A disk file explains how to make the changes, and after only a few minutes, I was able to change the main menu format from vertical to horizontal and to define the Ctrl-F10 keychord as a way to directly access the optimization menu. While there are many things you can change, there are others (such as an inability to extend the undo feature past one event) you have no control over. Once you've finished making changes, you can incorporate them into the TopSpeed environment by running the TSCFG.EXE program.

Although the editor handles up to ten windows (0–9), you'd normally edit in windows #1 through #9, because window #0 is a special window, called the "error editor window." This window comes into play after errors and warnings are found during compilation of a source file. The flawed file is displayed in this window with the cursor positioned at the first error, and the corresponding error message appears at the bottom of the window. Pressing F8 moves you forward to the location of the next error; F7, back to the previous one.

When you exit from the TopSpeed environment, the system remembers the contents and status of each window and reloads the same files the next time you enter the environment. You can alternatively start with a "clean slate" by supplying the */n* option on the command line. Another nice touch is a prompt reminding you to save your work when you call up the source-level visual interactive debugger (VID) from the TopSpeed menu.

There are a number of useful options available under the Utilities menu, including an ASCII table, a programmer's calculator that works in decimal, hex, or binary, and a window that lets you see the scan codes for keyboard keys. There is a multiple-file string search capability that works a bit such as *grep*, albeit without the powerful regular expression capabilities of that Unix utility. Other options include the ability to print files, to view files as data (that is, in hex), and to display system information. "System Info" shows the current date, time, and directory, the names of the files being edited in the TopSpeed windows, and a summary of free space on all disks. Be prepared to wait for this report if you have a CD-ROM disk attached to your system, for even though there is no "free space" available on a CD-ROM, there is no way to tell TopSpeed to ignore the drive, which gets interrogated in turn along with the other drives in your system.

There are a number of other features I found useful in this environment, too many in fact to list. Particularly noteworthy to me, however, are the ability to have up to nine generations of backup copies (I have mine set to 3), and the ability to record, load, save, and playback keystroke macros. The instructions for recording a macro were clear, and it took only a couple of minutes for me to create a macro that toggled all optimization on and off. About the only criticism I have of the editing environment is the lack of mouse support and the lack of Unix-style regular expression parsing (as in Brief, for example), but those are relatively minor annoyances.

Compiling

While the editor and its features form an important part of the TopSpeed C package, you can't forget that this is, after all, a C compiler, and the worth of the package ultimately hinges on how well it compiles code.

To help put TopSpeed C through its paces, I worked with a file that had been written in Microsoft C for a fairly simple-minded game of deduction. The playing "field" for this game is a 9 × 15 grid that contains some number of hidden mines, and the player uses the numeric keypad to "walk" a happy-face character through this mine field to a goal. To give the player a fighting chance, the number of mines in adjacent squares is displayed as the player moves from square to square, thus allowing the player to deduce the location of the mines without "stepping" on them. To make life easier, the location of mines may be marked by pressing M and an appropriate key on the numeric keypad. In addition, if an */s* parameter is passed on the command line when the program is invoked, the program won't let the player step on such marked squares. Finally, typing **?** at the start of the game causes the program to start playing by itself until it either gets to the goal or is unable to proceed further through the grid. Aside from being fun to play, the file KABOOOOM.C (see Listing One, page 109) is just under 1000 lines in length and offers a substantial chunk of source code for the compiler to process.

The first attempt to compile the code resulted in a couple of errors. The first error, in the function *DisplayCell*, had to do with a failure to read an embedded ASCII 2 (the happy-face) in the source code. The TopSpeed editor did not read this character, resulting in the assignment *Char = '';* and a subsequent error. Changing the line to *Char = 2;* fixed the problem.

The second error came in a line of the *DisplayChar* function, which read: *FP_SEG(cPos) = 0x0b000;*

While a legitimate statement in Microsoft C, this use of FP_SEG( ) generated a "left operand of assignment must be a modifiable lvalue" error message from the TopSpeed compiler. Pressing F1 for help brought up an explanation of the message, which is comfortably verbose as it is. I moved off the offending line and again pressed F1, and shortly was reading the help screen associated with FP_SEG( ). It referred me to MK_FP( ), which permitted me to replace the offending line (and the line after it) with: *cPos = MK_FP(0x0b800,((x+y*80)<< 1));* which eliminated the problem. A quick check showed that Turbo C 2.0 also required the MK_FP( ) syntax in order to compile without error. I later learned that the FP_SEG macro is defined differently for the Microsoft and TopSpeed compilers, which explains the failure to compile.

Once the bugs were corrected, the initial compilation pass with TopSpeed C took about 13 seconds on my 16-MHz ARC 386i computer, and optimization took another 18 seconds or so. With all optimizations turned off (there are nine forms of optimization, including optimization for time and space as well as constant, jump, peephole, loop, and alias optimization), the compile time was cut down to 28 seconds overall. This compared favorably with a run through Microsoft C 5.1, which took 37 seconds to compile without optimization, yet was slower than Turbo C 2.0, which compiled and linked the program in about 13 seconds.

I ran the compiler from within the editing environment, but you can also run TopSpeed C from the command line, where a comprehensive set of command-line options give the programer complete control of compilation and linking. In fact, there are four ways to set compiler options in the TopSpeed C compiler: From a menu, from the command line, via directives in TopSpeed's *make* facility, and by including pragmas in the source code.

One very topical option in the compiler is the ability to check for ANSI compatibility. JPI has made a point of maintaining 100 percent conformance to the ANSI C definition, and now that the seemingly interminable deliberations of the ANSI C committee have apparently come to a close, this feature should be a point in TopSpeed C's favor.

Making it With Project Files

TopSpeed's project files make it easy to admit to hating traditional make programs. Like its traditional counterpart, TopSpeed's Make uses a text file (called a "project file") to figure out what kind of file to produce with what objects and libraries, and using which memory model. Project files are collections of "directives" that, in addition to the usual specification of object modules, libraries, and so on, establish various compiler and linker options (such as inclusion of debugger information in the .EXE file), override options for specific files or groups of files, and specify what programs to run (if any) after the make process is complete. You could, for example, copy the .EXE file to a diskette every time you compiled and linked the source code. In short, the project file is the mechanism by which TopSpeed source code is transformed into executable files.

Two valuable features aid in the link process. Type-safe linking involves catching function calls made with the wrong parameter types. You will bless this feature the first time it saves you from calling an external function with the wrong parameter types. The technique of smart linking helps keep executable file size down and reduces the complexity associated with maintaining libraries. When you link a program, TopSpeed will only include those routines that are referenced in the code, leaving all the other routines out of the executable file. Strangely enough, this means that sometimes you must make an extraneous reference to a variable in order to make sure certain routines are linked into the .EXE file. A case in point is the need to include a line *includePMD = 1;* in one of the functions that handles critical program errors so that TopSpeed loads the appropriate routines to perform a post-mortem dump in case the program bombs.

Debugging

The VID is a full source-level symbolic debugger that uses overlapping windows in an interactive environment. Like the parent TopSpeed environment, there is no mouse support in VID.

VID is easy to run from the TopSpeed environment, which wisely prompts you to save your files before it swaps itself to disk, leaving room for your program and VID. To use VID, you need to generate VID information during compilation and a .MAP file during the link process. All required files are found using the redirection file, if necessary.

All the usual debugging features are here. You can set and clear breakpoints, create "sticky" breakpoints, examine different types of variables, find procedures, evaluate expressions, all the usual stuff. While not as powerful as, say, the Borland Turbo Debugger (there is no equivalent to the Inspect command, for example, which shows record structure, or the CPU window, which shows registers, memory, and disassembled code all at once) the VID is nevertheless a competent piece of software that is able to do the job.

TechKit

The TechKit is what distinguishes the Extended Edition ($395 list price) from the Standard Edition ($195) of TopSpeed C. What you get for your money is a collection of programs, files, and utilities that add functionality to TopSpeed C. It includes support for Windows programming and dynamic link libraries (DLLs), including DLLs that can be used under DOS. (A DLL is an OS/2 innovation that allows applications to share common data and code by linking library routines at run time.)

A major piece of the Techkit is the source code to the TopSpeed libraries. This collection of files fills over 1.5 Mbytes of disk space. The code is designed not only for use by TopSpeed C, but also for use with other TopSpeed languages. Many of the files are written in TopSpeed's assembler language, which makes for speedy routines, but also requires you to learn a new dialect of assembler.

The TopSpeed Assembler attempts to gain in simplicity and speed by deviating from "standard" 8086 assemblers in several ways. For example, the lexical structure of the assembler is derived from Modula-2, memory operands and segment overrides must always be explicitly stated, and there are no macros used in the language. While

I can understand JPI's reason for doing it this way, I don't look forward to becoming familiar with yet another assembler scheme.

An interesting utility included in the TechKit is WATCH, which lets you specify groups or individual DOS functions to monitor during program execution. I ran the program and specified the date/time functions for monitoring, with output to be sent to my printer (as opposed to the screen or disk file). When I ran KABOOOOM.EXE, a brief report was sent to the printer when the program called DOS to get the time during the initialization phase. The Alt-backspace keychord toggles WATCH on and off, and in order to change the scope of the DOS functions monitored, I found it necessary to unload WATCH and then reload it from scratch. (If you send WATCH's output to the screen, however, you are able to interact more with the program [setting and clearing functions to monitor], albeit at the expense of interfering greatly with the screen.) WATCH, of course, will not work if the program it is monitoring does not use DOS to accomplish its ends.

Other pieces of the TechKit aid in the debug and streamline process. The post-mortem debugger was undoubtedly created for those who've wished their bug-ridden programs could leave some indication behind them of what went wrong before they exit to never-never land. This feature is set by including the PMD.H file and referencing the *includePMD* variable in the source code. Should anything go wrong and a critical error function is called, your program creates a file that details the state of the system just before lights out. This file can be examined using the VID.

I've always gritted my teeth when sitting down to work with a profiler, but I found the TopSpeed TSPROF profiler easy to use. All you need to use TSPROF is a .MAP file, which is created when the program is made. I ran the profiler for KABOOOOM and found that over half the program's time is spent executing DOS routines, nearly half the program's time is spent in the BIOS, and only three percent or so of the time is in the code.

Conclusion

Working with the TopSpeed C compiler was a wholly pleasant experience. The advantage of having the file redirection and project file features are alone almost worth the price of admission, and the overall flexibility of the system is a big plus. Though it remains to be seen whether JPI will be able to successfully market the idea of a common programming environment with plug-in language modules, TopSpeed C certainly deserves to be a contender in the fight for a share of the C compiler market.

Acknowledgments

The author would like to thank Thomas D. Eldredge II for the use of his source code for KABOOOOM.C. Tom's program represents an enhancement of a game called RELENTLESS LOGIC by Conway, Hong, and Smith, which was found on the RBBS-IN-A-BOX CD-ROM.

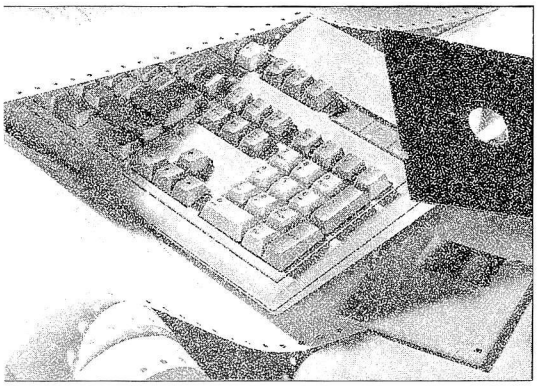
PRODUCT INFORMATION

TopSpeed C
Jensen & Partners International
1101 San Antonio Rd., Ste. 301
Mountain View, CA 94043
Price:
Standard Edition $199
Extended Edition $395
OS/2 Edition $495
Requirements: Extended Edition – IBM PC or compatible, DOS 2.0 or later, 640K RAM. Hard disk recommended.

DDJ

(Listing begins on page 109.)

Vote for your favorite feature/article.
Circle Reader Service **No. 7.**



Neural Networks and Image Processing

Finding edges only a human can see

Casimir C. "Casey" Klimasauskas

One of the key problems facing the machine vision industry is how to detect specific features in an image. It turns out that even finding a simple feature such as an edge can be difficult, if not impossible. Even though a person looking at a video camera image on a monitor can readily see the boundary between two objects, it may not be so easy to find it with an algorithm. Researchers studying how the eye preprocesses information for the brain use the term "early vision" for the function of the eye that assists in pattern recognition. We can use insights from research in early vision to solve the problem of edge detection by computer.

This article presents an engineering approximation of early vision, written from the perspective of an engineer investigating useful applications of neurally inspired technology. Although the techniques discussed here were suggested by the processes of the human eye, they are not intended to be biologically accurate, nor is the solution intended to be biologically plausible. The architecture of the edge detection system presented here is the empirical result of exploring many blind alleys and dead ends. For this reason, some of the assumptions and function values used here may seem somewhat arbitrary. Their only justification is that they worked.

The edge enhancement system presented here can be implemented in various ways, using different technologies. This article presents two implementations in software (one using the C language and the other using Lotus 1-2-3) and also describes a third implementation using commercially available image processing hardware and software.

Casimir C. "Casey" Klimasauskas is the founder of NeuralWare Inc., a supplier of neural-network development systems and services. Prior to that he worked extensively in machine vision and robotics. He can be reached at Penn Center West IV-227, Pittsburgh, PA 15276; 412-787-8222.

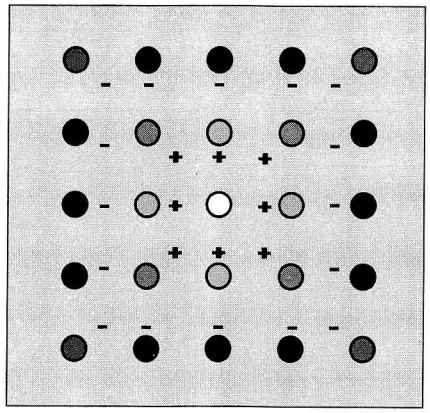


Figure 1: *Effect of a processing element on its neighbors. A "+" near a processing element indicates that the center processing element in the diagram will excite it if it is excited. A "–" near a processing element indicates it will be inhibited if the center processing element is excited*

A Logical Edge Enhancement Model

You might think of the receptive surface of the eye as an array or grid of photoreceptive elements. Light from the outside world impinges on this photoreceptive array and provokes output from each of the array elements. The output of each of these photoreceptors is passed on to another layer of corresponding neurons that work together to enhance the image.

For purposes of this article, we will call our two-layer network the edge enhancement system (EES). Figure 1 shows the effect of one of the EES processing elements. The connections are shown only from the processing element in the center of the array. This processing element excites its nearest neighbors (shown by "+" near the processing elements) and inhibits those a little further away (shown by " – " near the processing elements). The actual strength of the excitation or inhibition, as a function of distance from the center, is shown in Figure 2. When plotted in three-dimensions, with the magnitude of the excitation or inhibition as the Z-axis, the resulting shape looks like a Mexican hat. For this reason, it is sometimes called a "Mexican hat function" (MHF) or "on-center off-surround." The effect of the Mexican hat function is similar to that of a standard image processing filter known as a "difference of Gaussians."

The connections are shown only for the center processing element in Figure 1, all the other processing elements are connected in a similar fashion.

The EES processing element (shown in Figure 3) computes an internal activation value by computing the weighted sum of the outputs of its neighbors and the weights connecting them. This internal activation value is then transformed by a nonlinear transfer function (such as the clamped linear one shown) to produce an actual output. The clamped linear transfer function was found to work best after sigmoid and hyperbolic tangent transfer functions were tried and found not to work. Notice that the current output of a processing element is fed back onto itself as part of the input for computing its internal activation.

Readers familiar with neural-network types will recognize the EES array of processing elements described as a kind of feedback neural net, (similar to a Hopfield network, but with a fixed pattern of inter-connections). The connections are such that each processing element is trying to decide if it is on an edge or not. When this constraint is satisfied, the processing elements reach a stable output state.

In operation, the outputs of the receptor array are passed on to the EES. The initial values of each of the elements in the EES are equal to their corresponding values in the receptor array. After initialization, the EES goes through several iterations. During each iteration the processing elements obtain inputs from their neighbors (either excitatory or inhibitory) as well as from their current state. From these inputs, they compute a new output transformed through some nonlinear function. In the eye, these processes evolve as a dynamical system obeying a set of continuous differential equations defined by the synapses connecting them.

An EES Engineering Approximation

To develop a good engineering approximation, we need to be able to implement the EES inexpensively and efficiently. This section looks at techniques for accomplishing this with readily available off-the-shelf image processing hardware and software. The two principal image processing techniques discussed here are convolution and look-up tables.

Convolution is a common and powerful technique for filtering images. Very simply, a convolution is a specially designed matrix (or filter) that is combined together with a portion of an image to compute a transformed pixel value. The filter is centered at each pixel in the initial image and the "convolution" of the filter and the image beneath it is computed. The result is the transformed value of the center pixel. The matrix is then moved one pixel to the right and the transformed value of the next pixel is computed. When the filter has been applied, centered at each pixel in the initial image, the resulting transformed image is complete. This is shown in Figure 4.

The convolution of filter and image is arrived at by computing the pairwise product of corresponding elements of the filter and the underlying portion of the image and summing them together. Notice that this is the same as computing the internal activation of the EES processing element shown in Figure 3. This means we can implement the EES neural net by using standard image processing hardware that supports convolution.

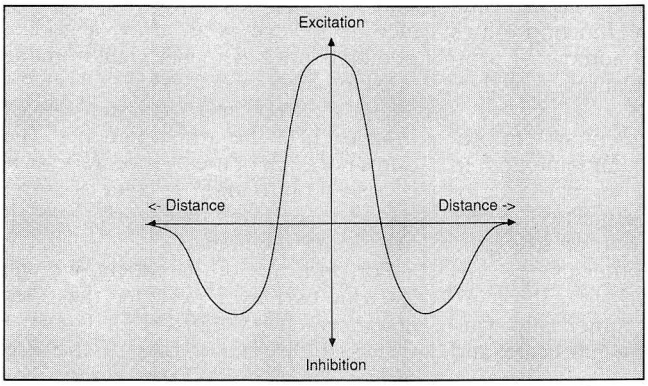


Figure 2: *Impact of a processing element on its neighbors. As the distance from the processing element in the center of the diagram increases, it excites its nearest neighbors. As the distance increases, the excitation turns to inhibition, and finally all effects cease. This curve is sometimes called a "Mexican Hat Function" or "Difference of Gaussians"*

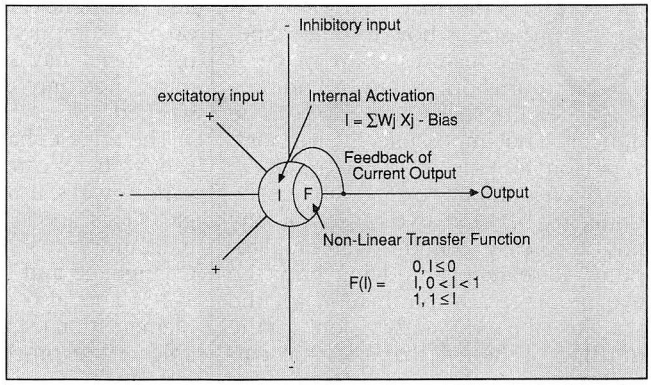


Figure 3: *Details of a single-processing element in the Edge Enhancement System. The "+" symbols indicate that the input will be excitatory. The "–" symbols indicate that the input will be inhibitory*

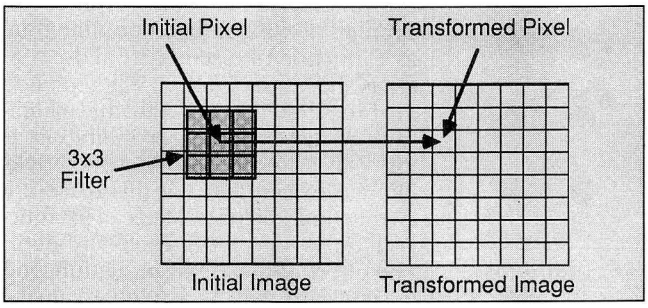


Figure 4: *Operation of a two-dimensional convolution. The 3 × 3 filter is laid over the initial image, centered at a particular pixel. The value of the matrix is multiplied by the pixel value just beneath it. The result of these pair-wise products are summed together to form the transformed pixel value in the transformed image*

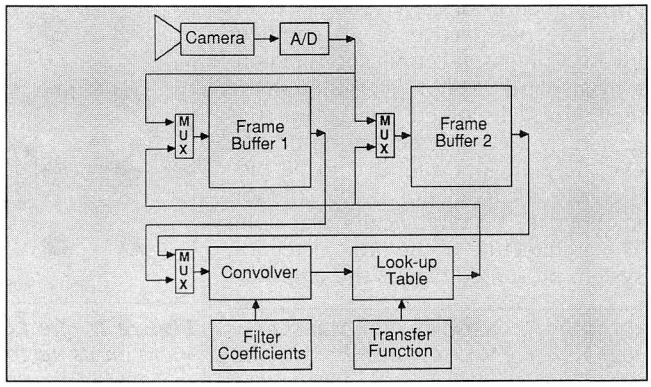


Figure 5: *Hardware block diagram of an image processing system adapted to be used to implement the Edge Enhancement System. The basic building blocks are a pair of frame buffers, a convolver, and a look-up table for performing nonlinear transformations*

Image filtering by use of convolutions is one of the cornerstones of machine vision. By properly selecting the coefficients of the filter, you can detect edges, create high- or low-pass filters, grow or shrink light regions, and quite a variety of other functions. You'll find more information on digital image filtering[2] at the end of this article. In practice, implementing an edge detector using a convolution filter is not difficult. The problem that arises is that of finding good filter coefficients, which do an effective job of finding the edges rather than losing or obscuring them.

A second commonly used technique in image processing is called a "look-up table." Just as the name implies, the value of a pixel is applied to the input of a look-up table (usually the address lines of a static RAM array) and a "transformed" value is produced at the output (the contents of that memory location). The mapping function is typically arbitrary and can be defined by the user.

Look-up tables are used to enhance contrast, convert images to black and white (from gray or color), and to produce special effects. The Cherry Coke commercials use this to make the can of Cherry Coke be in color and all else black and white. In our case, they can be used to implement a clamped linear transfer function. To implement a clamped linear transfer function in an 8-bit system, set the mapping RAM to output zero whenever an input in the range 0x80 through 0xff (negative values) is applied. For locations 0x00 through 0x7f, set the mapping RAM to output the same value as the input.

Implementing the EES

Both the convolution and look-up table techniques are such common tools that both are included in most commercial image processing systems. Together with a pair of frame buffers (also common), we can actually implement a very fast and moderately priced edge enhancement system. Companies that supply suitable hardware and software include Imaging Technologies (ITI), DataCube, Data Translation, and Matrox.

A block diagram of the hardware to implement the EES is shown in Figure 5. To set up the system, we load the block shown as "Filter Coefficients" with the coefficients from the MHF, and the look-up table "Transfer Function" with the values for a clamped linear transfer function; 7 × 7 is the minimum-sized convolution to use for the MHF. Some of the systems mentioned also support 9 × 9 and larger convolutions.

The sequence of processing is as follows:

1. Acquire an image from the camera to frame buffer 1.
2. Transform frame buffer 1 to frame buffer 2 using the MHF filter and clamped linear look-up table map function.
3. Transform frame buffer 2 to frame buffer 1 using the Mexican hat function filter and clamped linear look-up table map function.
4. Repeat steps 2 and 3 as many times as desired.

Because most systems are designed to work with small integers, it will be necessary to make the appropriate translations. This is an example of how neural-network technology can be grafted into existing technology to enhance its performance. With a little thought, it is possible to apply similar techniques to a variety of other problems.

Software Implementations of EES

When I began doing research on these filters for a project we are working on, I wanted something that would be easy to work with, and I could quickly try out a variety of parameters. After a little thought, I decided to try out my new

copy of Lotus 1-2-3. The spreadsheet instance described in this section is the result of those efforts. Though I used Lotus 1-2-3, Release 3.0, it should be possible to implement this with most spreadsheet packages and computers that support a graphing option.

As it turns out, a variety of other techniques could have also been used to do this research. Listing One, page 114, shows a C program that implements the same functions as the spreadsheet, but without the nice graphics or ability to change data as easily as with the spreadsheet. Both the C language implementation and the spreadsheet implementation deal with the more limited problem of a one-dimensional data stream rather than the two-dimensional image processing we have been discussing. Later in this article, I'll discuss how to extend the one-dimensional model to two-dimensions.

Listing Two, page 114, shows the spreadsheet constructed. The numbers to the right are the row numbers. The letters along the bottom represent the column numbers. The Graph capability of Lotus 1-2-3 is used to display the results from processing the one-dimensional signal or data stream. Although not every aspect of the spreadsheet is discussed here, the entire spreadsheet is available on-line or on disk from *DDJ*.

The first step in constructing the spreadsheet is to set up the "static" data. This consists of all titles, the "Bias" (cell D7), "Low Pass Filter" (range C20. .C28), "MHF Filter" (range D20. .D28), and "Raw Input Data" (range E16. .E124). Everything else in the spreadsheet is computed. This static data is entered exactly as shown. For the Raw Input Data, 0.00 represents "black" and 1.00 represents "white." Intermediate values may be used. Be careful to put everything in the cell locations shown. After the spreadsheet is constructed, you can move things around to suit your taste.

The calculations for the Low Pass Output data are as follows, assuming that you have entered the static data in the rows and columns shown. Enter the following equation in cell B20:

```
+$C$20*E16+$C$21*E17
                +$C$22*E18+$C$23*E19
                +$C$24*E20+$C$25*E21
                +$C$26*E22+$C$27*E23
                        +$C$28*E24
```

or with Lotus 1-2-3, Release 3:

```
@SUMPRODUCT
            ($C$20. .$C$28,E16. .E24)
```

Then replicate cell B20 throughout the range B21. .B120. This column is labeled as "Graph A" as a reminder of which graph range to use to display it. (Line 14 of the spreadsheet.)

Calculations for the neural-network filter are done in a single step. Compute the internal activation and transfer function as follows:

```
@MAX(0.0,@MIN(1.0,
                $D$20*E16+$D$21*E17
                +$D$22*E18+$D$23*E19
                +$D$24*E20+$D$25*E21
                +$D$26*E22+$D$27*E23
                    +$D$28*E24–$D$7))
```

or with Lottus 1-2-3, Release 3:

```
@MAX(0.0,@MIN(1.0,
            @SUMPRODUCT($D$20. .
              $D$28,E16. .E24)-$D$7))
```

Then replicate cell F20 throughout the range F21. .M120. The *@MAX(0, . .)* clamps the output so it can never go below zero. *@MIN(1, . .)* clamps the output so it can never go above one. The sum of the pair-wise products (or SUMPRODUCT) computes the effect of the neighborhood processing elements on the current one, and includes feedback of the current state. The *– $D$7* subtracts off the bias from the internal activation.

The first four and the last four cells in columns F through M are a copy of the values of the cells just prior to them. To replicate the values of the top of the columns, enter:

```
Cell F16: +F$20
```

Then replicate it throughout the range F16. .M19. To replicate the values at the bottom of the columns, enter:

```
Cell F121: +F$120
```

Then replicate it throughout the range F121. .M124. The computation portion of the spreadsheet is now complete. Use the graphing feature of your spreadsheet to construct the graphs described in Figure 6. These two graphs will be used to display the processing effects of various types of inputs and filters on the output data.

Testing the Spreadsheet Implementation

Having constructed the spreadsheet just described, the graph EES should look like the one in Figure 7a. Figure 7b is the same graph with the input range (Range B) reset, so it shows only the output of the network as it evolves. Figure 7c shows the input data and the final (eighth) iteration of the network with intermediate ranges reset (Ranges C, D, E).

The edge data for this experiment was selected to show profiles of two kinds of edges often found in images. In the first kind, light shines on a curved edge or rounded edge resulting in a gradation in intensities. The gradually changing light intensities on the left side of the graph are typical of this kind of edge. The second kind of edge is a ragged edge such as from torn metal. This type shows wide variations in gray level due to specular reflectivity as well as sharp variations in the curvature of the material. This is shown as the very noisy edge on the right of center of the diagram. Notice that the EES does a very nice job of sharpening both edges.

To the far right is a small "blip" in intensities. This blip is of the same magnitude as the one in the center of the main pulse. Notice that the EES was able to pick this out, because of its contrast to the background, while ignoring the noise on the top of the pulse. A little experimentation will show that this is quite a powerful technique. The bias value (in cell D7) can be changed to alter the sensitivity to various features. Changing the shape of the MHF also changes the nature of edges detected. Figure 8 shows the MHF used in the filter.

As it turns out, both edges used in this test tend to be difficult to find using standard image processing techniques. Figure 9 shows what happens when a simple sobel operator is applied to the input. The resulting derivative function does not provide much information about where the edges might be. The problem is not that such a filter is difficult to implement, but that finding a set of coefficients and a filter length, which enhances the edges rather than missing or obscuring them, is a highly heuristic and often frustrating task. My own experience is that it is sometimes impossible.

There are a variety of experiments you can do with the edge enhancement system. One of the things you

EES (Edge Enhancement System):

Format: Lines only

Graph	Range	Contents
B	E16. .E124	input data
C	F16. .F124	1st iteration
D	H16. .H124	3nd iteration
E	J16. .J124	5th iteration
F	M16. .M124	8th iteration

HIGHPASS (High Pass Filter):

Format: Lines only

Graph	Range	Contents
A	B20. .B120	Low-pass filtered data
B	E20. .E120	input data

Figure 6: *Constructing the graphs*

will discover is that the system can be sensitive to the shape of the MHF as well as the bias. In some ranges, the detected edges actually set up standing waves, which emanate out from the edges!

How It Works

As I mentioned at the beginning of this article, the EES is an engineering approach to image or signal processing based on biological insights. It was developed heuristically starting with a biological model and studying it until the mechanics of its operation were well understood. As such, there is no formal theory of operation.

Functionally what happens is that the MHF acts as a difference of Gaussians filter. The transfer function clips the negative part of the output, leaving only the positive center peak when the

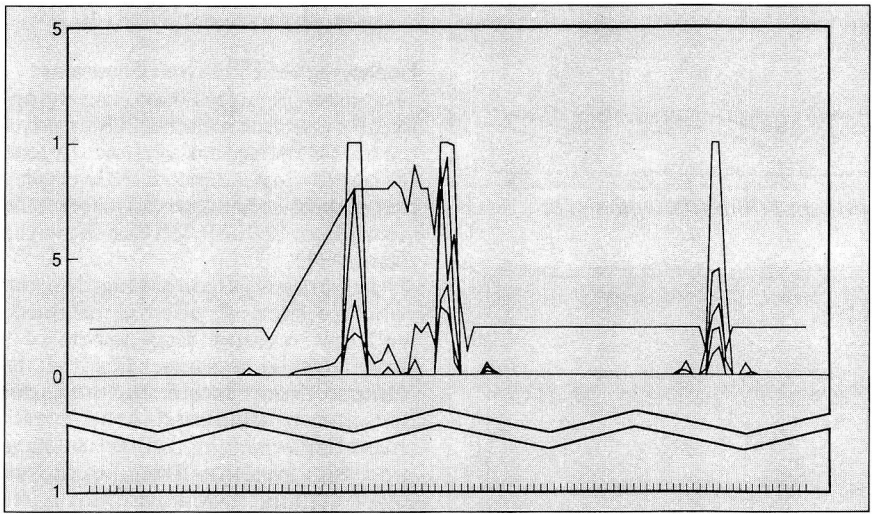


Figure 7a: *Output of graph EES of the Edge Enhancement System spreadsheet. The inputs, as well as the network outputs after 1, 3, 5, and 8 iterations, are shown super-imposed*

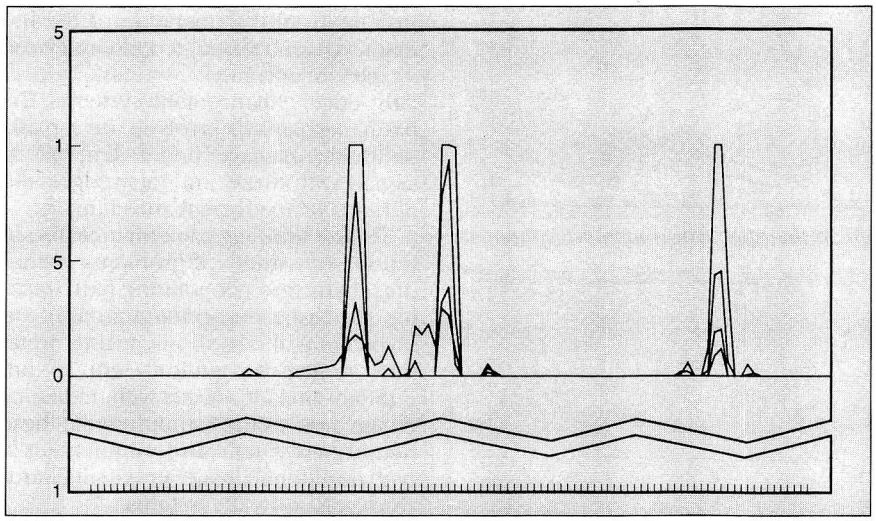


Figure 7b: *Output of the network as it evolves. This shows only the network outputs after 1, 3, 5, and 8 iterations*

```
 0.0 -0.1 -0.1 -0.2 -0.2 -0.2 -0.1 -0.1  0.0
-0.1 -0.5 -0.6 -0.6 -0.6 -0.6 -0.6 -0.5 -0.1
-0.1 -0.6 -0.4 -0.2 -0.2 -0.2 -0.4 -0.6 -0.1
-0.2 -0.6 -0.2  1.0  1.0  1.0 -0.2 -0.6 -0.2
-0.2 -0.6 -0.2  1.0  1.8  1.0 -0.2 -0.6 -0.2
-0.2 -0.6 -0.2  1.0  1.0  1.0 -0.2 -0.6 -0.2
-0.1 -0.6 -0.4 -0.2 -0.2 -0.2 -0.4 -0.6 -0.1
-0.1 -0.5 -0.6 -0.6 -0.6 -0.6 -0.6 -0.5 -0.1
 0.0 -0.1 -0.1 -0.2 -0.2 -0.2 -0.1 -0.1  0.0
```

Example 1: *A two-dimensional version of the EES*

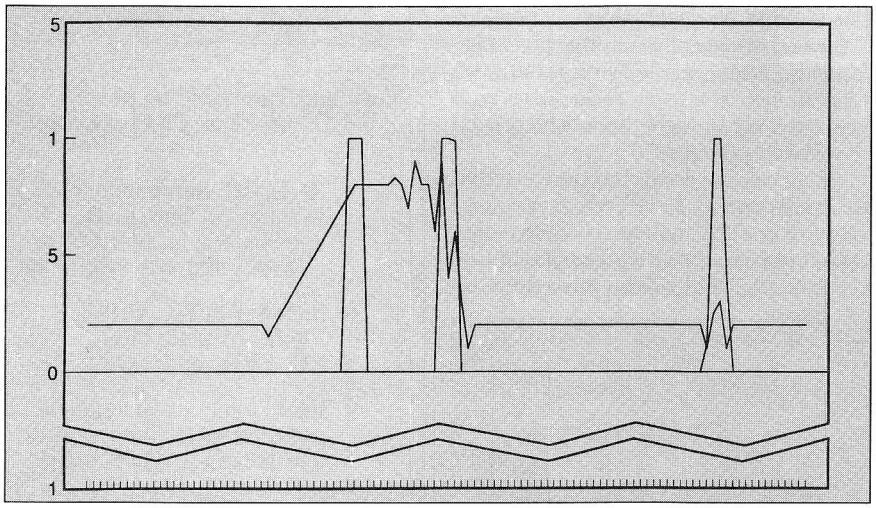


Figure 7c: *The original input to the network, and the output after eight iterations are shown super-imposed*

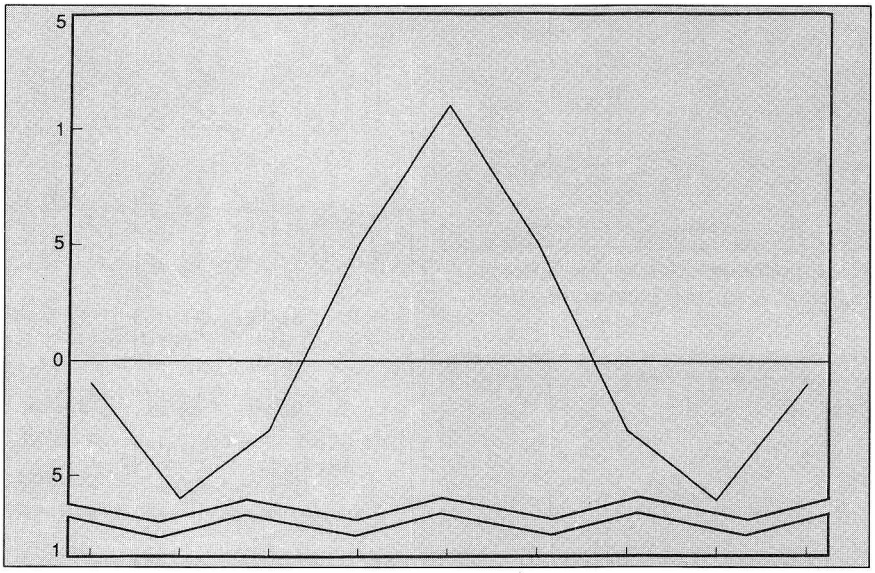


Figure 8: *The Mexican Hat Function used in the Edge Enhancement System*

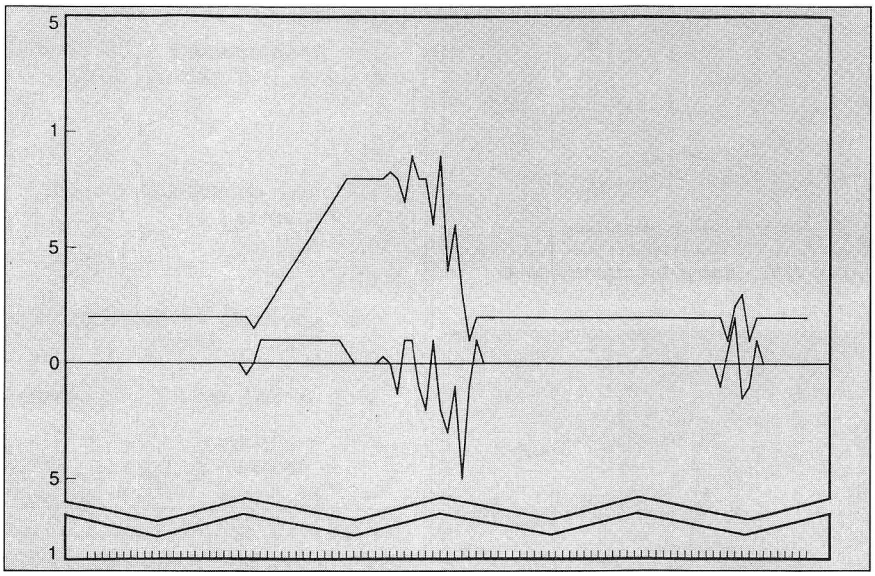


Figure 9: *Results of applying a sobel (simple edge detection) filter to the input data. The sobel computes the derivative of the image*

filter is directly over the edge. When the MHF is applied again to the resulting output, it will tend to enhance single peaks but reduce plateaus. As such, lots of noise in the vicinity of an edge will be ignored. However, a single substantial variation against a constant background will be significantly enhanced. Iterating on this process eventually results in groups of saturated processing elements, at most the width of the excitatory part of the MHF. All other processing elements are turned off.

Extending the EES to Two-Dimensions

The same principles used in developing the one-dimensional EES apply to the two-dimensional version. Instead of using a single-dimensional vector, a two-dimensional matrix is used. One example of a 9×9 MHF is shown in Example 1.

The process of computing the convolution (sum of pair-wise products) with the corresponding portion of a pixel array is the same. Likewise, the clamped linear transfer function uses the same equation used in the spreadsheet. Implementing this on an image processing system will require converting everything to work with small integers, but the process is quite straightforward.

Summary

Insights from the operation of the human eye can be used to build improved image enhancement systems, particularly edge enhancement systems. The basic mechanisms involved are capable of turning an image (or one-dimensional signal) with fuzzy and noisy edges into a sharp clean edge-enhanced image.

This technology can enhance the solution of a variety of problems including character recognition, part tracking, part inspection, printed circuit board inspection, ultrasonic image interpretation, target recognition, and so on. Enough similarities exist with traditional image processing techniques that these neural networks can be implemented with traditional image processing hardware and software systems.

References

1. Carver, Mead. *Analog VLSI and Neural Systems,* Addison-Wesley, Reading, Mass.: 1989.

2. Tzay Y. Young, King-Sun Fu. *Handbook of Pattern Recognition and Image Processing,* Academic Press, Orlando, Fla.: 1986.

DDJ

(Listings begin on page 114.)

Vote for your favorite feature/article.
Circle Reader Service **No. 8.**

Listing One *(Text begins on page 16.)*

```
////////////////////////////////////////////////////////
// BAM.HPP  Provide vector, matrix, vector pair, matrix, BAM matrix, and
// BAM system classes and methods to implement BAM system concept.
// Extended note:
// This is an implementation of the concept of Bidirectional
// Associative Memories as developed by Bart Kosko and others.
// It includes the extended concept introduced by Patrick Simpson
// of the "BAM System". Where reasonable Simpson's notation has been
// been maintained. The presentation benefits from C++ and OOP, in that
// (I believe) it is both easier to understand than a "pseudocode" version,
// yet more precise (in that it works!)
// Developed with Zortech C++ Version 2.0 -- Copyright (c) Adam Blum, 1989,90

#include<stdlib.h>
#include<io.h>
#include<stdio.h>
#include<string.h>
#include<limits.h>
#include<ctype.h>
#include<stream.hpp>
#define D(x) {if (trace){x}}
extern int trace;

// where are Zortech's min,max?
#define max(a,b)        (((a) > (b)) ? (a) : (b))
#define min(a,b)        (((a) < (b)) ? (a) : (b))

// will be changed to much higher than these values
const ROWS=16;  // number of rows (length of first pattern)
const COLS=8;   // number of columns (length of second pattern)
const MAXMATS=10; // maximum number of matrices in BAM system
const MAXVEC=16; // default size of vectors

class matrix;
class bam_matrix;
class vec {
        friend class matrix;
        friend class bam_matrix;
        friend class bam_system;
                int n;
                int *v;
        public:
                // see BAM.CPP for implementations of these
                vec(int size=MAXVEC,int val=0); // constructor
                ~vec(); // destructor
                vec(vec &v1); // copy-initializer
                int length();
                vec& operator=(const vec& v1); // vector assignment
                vec& operator+(const vec& v1); // vector addition
                vec& operator+=(const vec& v1); // vector additive-assignment
                vec& operator*=(int i); // vector multiply by constant
                // supplied for completeness, but we don't use this now
                int operator*(const vec& v1); // dot product
                vec operator*(int c); // multiply by constant
                // vector transpose multiply needs access to v array
                int operator==(const vec& v1);
                int& operator[](int x);
                friend istream& operator>>(istream& s,vec& v);
                friend ostream& operator<<(ostream& s, vec& v);
}; //vector class

class vecpair;

class matrix {
        protected:
        // bam_matrix (a derived class) will need to use these members
        // preferred to "friend class", since there may be many derived
        // classes which need to use this
                int **m; // the matrix representation
                int r,c; // number of rows and columns
        public:
                // constructors
                matrix(int n=ROWS,int p=COLS);
                matrix(const vec& v1,const vec& v2);
                matrix(const vecpair& vp);
                matrix(matrix& m1); // copy-initializer
                ~matrix();
                int depth();
                int width();
                matrix& operator=(const matrix& m1);
                matrix& operator+(const matrix& m1);
                matrix& operator+=(const matrix& m1);
                vec colslice(int col);
                vec rowslice(int row);
                friend ostream& operator<<(ostream& s,matrix& m1);
}; // matrix class

class vecpair {
        friend class matrix;
        friend class bam_matrix;
        friend class bam_system;
                int flag; // flag signalling whether encoding succeeded
                vec a;
                vec b;
        public:
                vecpair(int n=ROWS,int p=COLS); // constructor
                vecpair(const vec& A,const vec& B);
                vecpair(const vecpair& AB); // copy initializer
                ~vecpair();
                vecpair& operator=(const vecpair& v1);
                int operator==(const vecpair& v1);
                friend istream& operator>>(istream& s,vecpair& v);
                friend ostream& operator<<(ostream& s,vecpair& v);
                friend matrix::matrix(const vecpair& vp);
};

class bam_matrix: public matrix {
        private:
                int K; // number of patterns stored in matrix
                vecpair *C; // actual pattern pairs stored
                int feedthru(const vec&A,vec& B);
                int sigmoid(int n); // sigmoid threshold function
        public:
                bam_matrix(int n=ROWS,int p=COLS);
                ~bam_matrix();
                // but we supply it with the actual matrix A!B (W is implied)
                void encode(const vecpair& AB); // self-ref version
                // uncode only necessary for BAM-system
                void uncode(const vecpair& AB); // self-ref version
                vecpair recall(const vec& A);
                int check();
                int check(const vecpair& AB);
                // Lyapunov energy function: E=-AWBtranspose
                int energy(const matrix& m1); // Lyapunov energy function
}; // BAM matrix

class bam_system {
                bam_matrix *W[MAXMATS];
                int M; // number of matrices
        public:
                bam_system(int M=1);
                ~bam_system();
                void encode(const vecpair& AB);
                vecpair& recall(const vec& A);
                // train equiv. to Simpson's encode of all pairs
                void train(char *patternfile);
                friend ostream& operator<<(ostream& s,bam_system& b);
}; // BAM system class
```

End Listing One

Listing Two

```
////////////////////////////////////////
// BAM.CPP Provide vector, matrix, vector pair, matrix, BAM matrix, and BAM
// system classes to implement BAM systems
// Extended note:
// This is an implementation of the concept of Bidirectional
// Associative Memories as developed by Bart Kosko and others.
// It includes the extended concept introduced by Patrick Simpson
// of the "BAM System". Where reasonable Simpson's notation has been
// been maintained. The presentation benefits from C++ and OOP, in that
// (I believe) it is both easier to understand than a "pseudocode" version,
// yet more precise (in that it works!)
// Developed with Zortech C++ Version 2.0 -- Copyright (c) 1989,90 Adam Blum

#include"bam.hpp"

////////////////////////////////////
// vector class member functions

vec::vec(int size,int val) {
        v = new int[size];
        n=size;
        for(int i=0;i<n;i++)
                v[i]=0;
} // constructor
vec::~vec() { delete v;} // destructor
vec::vec(vec& v1) // copy-initializer
{
        v=new int[n=v1.n];
        for(int i=0;i<n;i++)
                v[i]=v1.v[i];
}
vec& vec::operator=(const vec& v1)
{
        delete v;
        v=new int[n=v1.n];
        for(int i=0;i<n;i++)
                v[i]=v1.v[i];
        return *this;
}
vec& vec::operator+(const vec& v1)
{
        vec sum(v1.n);
        sum.n=v1.n;
        for(int i=0;i<v1.n;i++)
                sum.v[i]=v1.v[i]+v[i];
        return sum;
}
vec& vec::operator+=(const vec& v1)
{
        for(int i=0;i<v1.n;i++)
                v[i]+=v1.v[i];
        return *this;
}
vec vec::operator*(int c)
{
        vec prod(length());
        for(int i=0;i<prod.n;i++)
                prod.v[i]=v[i]*c;
        return prod;
}
int vec::operator*(const vec& v1) // dot-product
{
        int sum=0;
        for(int i=0;i<min(n,v1.n);i++)
                sum+=(v1.v[i]*v[i]);
        //D(cout << "dot product " << *this << v1 << sum << "\n";)
        return sum;
}
int vec::operator==(const vec& v1)
{
        if(v1.n!=n)return 0;
        for(int i=0;i<min(n,v1.n);i++){
                if(v1.v[i]!=v[i]){
                        return 0;
                }
        }
        return 1;
```

```
}
int& vec::operator[](int x)
{
        if(x<length() && x>=0)
                return v[x];
        else
                cout << "vec index out of range";
}
int vec::length(){return n;} // length method

istream& operator>>(istream& s,vec &v)
// format: list of ints followed by ','
{
        char c;
        v.n=0;
        v.v=new int[MAXVEC];
        for(;;){
                s>>c;
                if(s.eof())return s;
                if(c==',')return s;
                if(isspace(c))continue;
                v.v[v.n++]=((c!='0')?1:-1);
        }
}
ostream& operator<<(ostream& s, vec& v)
// format: list of ints followed by ','
{
        for(int i=0;i<v.n;i++)
                s << (v.v[i]<0?0:1);
        s << ",";
        return s;
}

//////////////////////////////
// matrix  member functions
matrix::matrix(int n,int p)
{
        //D(cout << "Constructing " << n << " x " << p << " matrix.\n";)
        m=new int *[n];
        for(int i=0;i<n;i++){
                m[i]=new int[p];
                for(int j=0;j<p;j++)
                        m[i][j]=0;
        }
        r=n;
        c=p;
} // constructor
matrix::matrix(const vecpair& vp)
{
        //D(cout << "Constructing matrix from: " << vp;)
        r=vp.a.length();
        c=vp.b.length();
        m=new int *[r];
        for(int i=0;i<r;i++){
                m[i]=new int[c];
                for(int j=0;j<c;j++)
                        m[i][j]=vp.a.v[i]*vp.b.v[j];
        }
}// constructor
matrix::matrix(const vec& v1,const vec& v2)
{
        //D(cout << "Constructing matrix from " << v1 << v2 << "\n";)
        r=v1.length();
        c=v2.length();
        m=new int *[r];
        for(int i=0;i<r;i++){
                m[i]=new int[c];
                for(int j=0;j<c;j++)
                        m[i][j]=v1.v[i]*v2.v[j];
        }
}// constructor
matrix::matrix(matrix& m1) // copy-initializer
{
        //D(cout << "matrix copy-initializer\n"; )
        r=m1.r;
        c=m1.c;
        m=new int *[r];
        for(int i=0;i<r;i++){
                m[i]=new int[c];
                for(int j=0;j<c;j++)
                        m[i][j]=m1.m[i][j];

        }
}
matrix::~matrix()
{
        for(int i=0;i<r;i++)
                delete m[i];
        delete m;
} // destructor
matrix& matrix::operator=(const matrix& m1)
{
        for(int i=0;i<r;i++)
                delete m[i];
        r=m1.r;
        c=m1.c;
        m=new int*[r];
        for(i=0;i<r;i++){
                m[i]=new int[c];
                for(int j=0;j<r;j++)
                        m[i][j]=m1.m[i][j];
        }
        return *this;
}
matrix& matrix::operator+(const matrix& m1)
{
        matrix sum(r,c);
        for(int i=0;i<r;i++)
                for(int j=0;j<r;j++)
                        sum.m[i][j]=m1.m[i][j]+m[i][j];
        return sum;
```

(Listing continued on page 86)

Listing Two *(Listing continued, text begins on page 16.)*

```
}
matrix& matrix::operator+=(const matrix& ml)
{
        //D(cout << "matrix additive assignment\n";)
        for(int i=0;i<r&&i<ml.r;i++)
                for(int j=0;j<c&&j<ml.c;j++)
                        m[i][j]+=(ml.m[i][j]);
        return *this;
}
vec matrix::colslice(int col)
{
        vec temp(r);
        for(int i=0;i<r;i++)
                temp.v[i]=m[i][col];
        return temp;
}
vec matrix::rowslice(int row)
{
        vec temp(c);
        for(int i=0;i<c;i++)
                temp.v[i]=m[row][i];
        return temp;
}
int matrix::depth(){return r;}
int matrix::width(){return c;}

ostream& operator<<(ostream& s,matrix& ml)
// print a matrix
{
        for(int i=0;i<ml.r;i++){
                for(int j=0;j<ml.c;j++)
                        s << ml.m[i][j] << " ";
                s << "\n";
        }
}
////////////////////////////////////////
// vecpair  member functions
// constructor
vecpair::vecpair(int n,int p) { }
vecpair::vecpair(const vec& A,const vec& B) {a=A;b=B;}
vecpair::vecpair(const vecpair& AB) {*this=vecpair(AB.a,AB.b);}
vecpair::~vecpair() {} // destructor
vecpair& vecpair::operator=(const vecpair& v1)
{
        a=v1.a;
        b=v1.b;
        return *this;
}
int vecpair::operator==(const vecpair& v1)
{
        return  (a == v1.a) && (b == v1.b);
}
istream& operator>>(istream& s,vecpair& v1)
// input a vector pair
{
        s>>v1.a>>v1.b;
        return s;
}
ostream& operator<<(ostream& s,vecpair& v1)
// print a vector pair
{
        return s<<v1.a<<v1.b<<"\n";
}
////////////////////////////////
//bam_matrix  member functions
bam_matrix::bam_matrix(int n,int p):(n,p)
{
        // the maximum number of pattern pairs storable
        // is around min(n,p) where n and p are
        // the dimensionality of the matrix
        C=new vecpair[min(n,p)*2];
        K=0;
}
bam_matrix::~bam_matrix()
{
} // destructor
void bam_matrix::encode(const vecpair& AB)
// encode a pattern pair
{
        //D(cout << "BAM Matrix encoding: " << AB;)
        matrix T(AB);
        (*this)+=T; // add the matrix transpose to the current matrix
        C[K]=AB;
        K++;
}
void bam_matrix::uncode(const vecpair& AB)
// get rid of a stored pattern (by encoding A-B complement)
{
        //D(cout << "uncode\n";)
        vec v=AB.b*-1;
        matrix T(AB.a,v); // T is A transpose B complement
        *this+=T;// add the matrix transpose to the current matrix
        K--;
}
vecpair bam_matrix::recall(const vec& A)
// BAM Matrix recall algorithm (used by BAM SYSTEM recall)
{
        int givenrow=(A.length()==width());
        D(cout<<"BAM matrix recall of" << A << givenrow?"(row)\n":"(col)\n";)
        vec B(givenrow?depth():width(),1);
        for(;;){ // feed vectors through matrix until "resonant" pattern-pair
                feedthru(A,B);
                if(feedthru(B,A))break; // stop when returned A = input A
        }
        D(cout<< "resonant pair " << A << "\n and " << B << "\n";)
        if(givenrow)
                return vecpair(B,A);
        else
                return vecpair(A,B);
}
int bam_matrix::feedthru(const vec&A,vec& B)
{
        //D(cout << "Feeding " << A << "\n"; )
        vec temp=B;int n;
        for(int i=0;i<B.length();i++){
                if(A.length()==width())
                        n=sigmoid(A*rowslice(i));
                else
                        n=sigmoid(A*colslice(i));
                if(n)
                        B.v[i]=n;
        }
        return B==temp;
}
int bam_matrix::sigmoid(int n)
// VERY simple (but classic one for BAM) threshold function
//
//            1 --------------
//              |
//  - -----------           +
//           -1
{
        if(n<0)return -1;
        if(n>0)return 1;
        return 0;
}
int bam_matrix::check()
// check to see if we have successfully encoded pattern-pair into this matrix
{
        D(cout << "Check BAM matrix for " << K << " pattern pairs\n";)
        vecpair AB;
        for(int i=0;i<K;i++){
                AB=recall(C[i].a);
                if(!(AB==C[i])){
                        D(cout <<"failed check\n ";)
                        return 0;
                }
        }
        D(cout << "passed check\n ";)
        return 1;
}
int bam_matrix::check(const vecpair& AB)
{
        // different check routine for orthogonal construction BAM
        //check to see energy of present pattern pair to matrix
        // is equal to orthogonal BAM energy
        matrix T(AB);
        return energy(T)== -depth()*width();
}
int bam_matrix::energy(const matrix& ml)
{
        int sum=0;
        for(int i=0;i<depth();i++)
                for(int j=0;j<width();j++)
                        sum+=(ml.m[i][j]*this->m[i][j]);
        D(cout << "Energy of matrix " << -sum << "\n";)
        return -sum;
}

////////////////////////////////////////
// bam system  functions
// top level of system (for now)

// constructor
bam_system::bam_system(int n)

{
        M=n;
        for(int i=0;i<M;i++)
                W[i]=new bam_matrix;
}
bam_system::~bam_system() // destructor
{
        for(int i=0;i<M;i++)
                delete W[i];
}
void bam_system::encode(const vecpair& AB)
// encode the pattern pair AB into the BAM system
{
        D(cout << "BAM System encode\n";)
        for(int h=0;h<M;h++){
                W[h]->encode(AB);
                if(!W[h]->check())
                        W[h]->uncode(AB);
                else
                        break;
        }
        if(h==M){ // all matrices full, add another
                if(h<MAXMATS){
                        W[M]=new bam_matrix();
                        W[M]->encode(AB);
                        M++;
                }
                else{
                        cout << "BAM System full\n";
                        exit(1);
                }
        }
}
vecpair& bam_system::recall(const vec& A)
// presented with pattern A, recall will return pattern-PAIR
{
        vecpair XY[MAXMATS];matrix *M1,*M2;
        int E,minimum=0,emin=INT_MAX;
        D(cout << "BAM System recall\n";)
        for(int h=0;h<M;h++){
                XY[h]=W[h]->recall(A);
                D(cout << h <<"-th matrix, returned vecpair "<< XY[h];)
                M1=new matrix(XY[h]);
```

(Listing continued on page 88)

Listing Two *(Listing continued, text begins on page 16.)*

```
                E=W[h]->energy(*M1);
                if(A.length()==W[h]->width())
                        M2=new matrix(XY[h].a,A);
                else
                        M2=new matrix(A,XY[h].b);
                if (  ( E-(W[h]->depth()*W[h]->width()) < emin )
                   && (E==W[h]->energy(*M2))
                )
                {
                        emin=E-(W[h]->depth()*W[h]->width());
                        minimum=h;
                }
                delete M1;
                delete M2;
        }
        return XY[minimum];
}
void bam_system::train(char *patternfile)
// A "multiple-pair" encode - which Simpson calls "encode"
// this could be used for initial BAM Sys training. However an up
// and running BAM Sys should only need to use "encode".
{
        FILE *f=fopen(patternfile,"r");int n=0;
        filebuf sfile(f);
        istream s(&sfile,0);
        vecpair AB;
        for(;;){
                s >> AB;
                if(s.eof())break;
                D(cout << "Encoding " << n++ << "-th pattern pair:\n" << AB;)
                encode(AB);
        }
        D(cout << "Completed training from " << patternfile;)
}
ostream& operator<<(ostream& s,bam_system& b)
// operator to print out contents of entire BAM system
{
        for(int i=0;i<b.M;i++)
                s<< "BAM Matrix " << i << ": \n" << *(b.W[i]) << "\n";
}
```

End Listing Two

Listing Three

```
//////////////////////
// TESTBAM.HPP
// Interactive BAM System Demonstration Program. Used to verify BAM system
// algorithms and demonstrate them on an abstract (i.e. just 0s and 1s) case.
// Developed with Zortech C++ 2.0 -- Copyright (c) 1989,90 Adam Blum

#include"bam.hpp"

vec v;
vecpair AB;
bam_system B;
char *p;
char patternfile[16]="TEST.FIL"; // file where test data is stored
int trace=0; // SET TRACE=<whatever> at DOS prompt to turn trace on
main()
{
     cout << "Interactive BAM System Demonstration\n";
     trace=(p=getenv("TRACE"))?1:0;
     cout << "Training from " << patternfile << "\n";
     B.train(patternfile);
     D(cout << "Resulting BAM System\n" << B;)
     cout <<"Enter patterns as 0's and 1's terminated by comma.\n"
     <<"Patterns must be length of " << ROWS << " or " << COLS <<".\n"
     << "Null vector (just "","") to end.\n\n" ;
     for(;;){
          cout << "Enter pattern: ";
          cin >> v;
          if(!v.length())break;
          if(v.length()!=ROWS && v.length()!=COLS){
               cout << "Wrong length.\n";
               continue;
          }
          AB=B.recall(v);
          cout << "Recalled pattern pair\n" << AB;
     }
}
```

End Listing Three

Listing Four

```
1100101011010011,11101010,
0110110111110110,11010101,
1101111001010101,11110010,
1010101000010111,11001101,
0011001101011011,11110100,
1100101011010011,11101010,
0110100111110110,11010101,
1101110101010101,11110010,
1011101010010111,11001101,
0001011101011011,11110100,
1100101001010011,11101010,
0110110110110110,11010101,
1100111011010101,11110011,
1010000100010111,11001101,
0001101101011011,11110110,
1100100011010011,11100110,
0110110011110110,11010101,
1101111001010101,11110011,
1010100000011111,11001101,
0001100101111011,11111000,
1100101011010011,11011010,
0010100111110110,11010101,
1101111101010101,11110010,
1010111000010111,11101101,
0001000001011011,11110100,
1100101011010011,11101010,
0110110111110110,11010101,
1101111000010101,11110110,
1010100111010111,11001101,
0001000101011011,11110100,
0110110101110110,11010111,
1101111001010101,11110110,
1010111100110111,11001101,
0001000101011011,11110100,
1100101010010011,11101010,
0110110111110110,11010101,
1101111001010101,11110010,
1010110000010111,11001101,
0011000101011011,11110100,
0011010101111011,10010111,
```

End Listing Four

Listing Five

```
# TESTBAM.MK
# Make file for BAM System implementation tester
# Uses Microsoft Make
# Compiler: Zortech C++ 2.0
# To make with diagnostics enabled:
# make CFLAGS="-DDEBUG=1" testbam.mk
#

CFLAGS=
.cpp.obj:
     ztc -c $(CFLAGS) $*.cpp
bam.obj: bam.cpp bam.hpp
testbam.obj: testbam.cpp bam.hpp
testbam.exe: testbam.obj bam.obj
     blink testbam bam;
```

End Listings

Listing One *(Text begins on page 28.)*

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: andy; Base: 10 -*-

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;       A Research Environment for the Instantiation of Neural Networks
;;;       Andrew  J. Czuchry, Jr. -- Georgia Institute of Technology
;;;       Georgia Tech Research Institute -- Artificial Intelligence Branch
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;            Knowledge Representation Structure definitions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defstruct (NET                              ; structure for a network
            (:print-function net-printer))     ; printer function
    layers                   ; list of layers
    local-parameters         ; list of local parameters for net
  )
(defstruct (LAYER                            ; structure for a layer
            (:print-function layer-printer))   ; printer function
  planes                   ; list of planes
  e-connections            ; list of offsets for excitatory connections
  i-connections            ; list of offsets for inhibitory connections
  local-parameters         ; list of local parameters for layer
  prev-layer               ; "ptr" to preceding layer structure
  type                     ; layer type (e.g., "S" or "C" in
                           ;  neocognitron; "F1" or "F2" in ART)
  )
(defstruct (PLANE            ; structure for a plane (sub-network)
            (:print-function plane-printer))  ; printer function
  (cells nil :type array)    ; cell values for plane
  e-connections              ; list of offsets for excitatory connections
  i-connections              ; list of offsets for inhibitory connections
  e-weights                  ; list of excitatory weights [real]
                             ;  (same order as list of connections)
  i-weights                  ; list of inhibitory weights [real]
                             ;  (same order as list of connections)
  size                       ; size of plane (list N x M)
  local-parameters           ; list of local parameters for plane
  layer                      ; "ptr" back to layer of which plane is a part
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;     Structure Printer Functions -- Written by Harold S. Forbes

;;; ---------------------------------------------------------------------------
;;; The function OBJECT-ADDRESS gets the memory address of any LISP object.

(defun OBJECT-ADDRESS (object)
  ;; Symbolics implementation.
  (sys:%pointer object)
  )
(defun NET-PRINTER (structure stream ignore)
  (declare (ignore ignore))
  (format stream "#<net ~X>" (object-address structure)))
(defun LAYER-PRINTER (structure stream ignore)
  (declare (ignore ignore))
  (format stream "#<~A-layer ~X>" (layer-type structure)
          (object-address structure)))
(defun PLANE-PRINTER (structure stream ignore)
  (declare (ignore ignore))
  (format stream "#<plane ~X>" (object-address structure)))
```

End Listing One

Listing Two

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: andy; Base: 10 -*-

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;       A Research Environment for the Instantiation of Neural Networks
;;;       Andrew  J. Czuchry, Jr. -- Georgia Institute of Technology
;;;       Georgia Tech Research Institute -- Artificial Intelligence Branch
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;                          Net CREATION functions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Create a new net.
;   Creates a version of a 5-layer neocognitron by default.

(defun CREATE-NET (&key
                   (num-of-layers 5)
                   (num-of-planes-per-layer-list '(1 24 24 24 24))
                   (plane-size-list '((16 16) (16 16) (10 10) (4 4)
                                      (1 1)))
                   (connection-pattern 'square)
                   (mask-size-list '((5 5) (5 5) (5 5) (2 2)))
                   (net-parameters '(:  0.5))
                   (additional-parameter-list
                    `(:net-type neocognitron
                                (:r-val-list '(4.0 1.5 1.5))
                                (:q-val-list '(1.0 16.0 16.0))
                                (:b-val-list '(0.0 0.0 0.0))
                                (:orientation-list '(8 1 8 1))
                                )))
  (let* ((layer-list
          (create-layer-list num-of-layers
                             num-of-planes-per-layer-list
                             plane-size-list
                             connection-pattern
                             mask-size-list
                             additional-parameter-list))  ; create layers
                                                          ;  meeting specification
                                                          ;  parameters
         (net (make-net :local-parameters net-parameters
                        :layers layer-list)))   ; create knowledge structure
                                                ; storing new net
    net)
  )

;; Create a list of NUM-OF-LAYERS layers of the approriate type (type
;;   key recorded in ADDITIONAL-PARAMETER-LIST).
(defun CREATE-LAYER-LIST (num-of-layers planes-per-layer-list
                          plane-size-list con-pattern
                          mask-size-list
                          additional-parameter-list)
  (let ((layer-type (zl:lexpr-funcall #'extract-type-key
                                      additional-parameter-list)))
; determine type of net to which layers are to belong
;  and create appropriate type of layers
    (zl:selectq  layer-type
      (neocognitron
        (create-neocognitron-layer-list
          num-of-layers planes-per-layer-list
          plane-size-list con-pattern mask-size-list
          additional-parameter-list))
      (ART2
        (create-ART2-layer-list
          num-of-layers planes-per-layer-list
          plane-size-list con-pattern mask-size-list
          additional-parameter-list))
      (backpropagation
        (create-backprop-layer-list
          num-of-layers planes-per-layer-list
          plane-size-list con-pattern mask-size-list
          additional-parameter-list))
      )
    )
  )

;; Create a list of NUM-OF-LAYERS layers for the neocognitron
(defun CREATE-NEOCOGNITRON-LAYER-LIST (num-of-layers planes-per-layer-list
                                                     plane-size-list
                                                     con-pattern
                                                     mask-size-list
                                                     additional-parameter-list)
; extract parameters specific to neocognitron
  (let* ((r-val-list (zl:lexpr-funcall #'extract-r-val-list
                                       additional-parameter-list))
         (q-val-list (zl:lexpr-funcall #'extract-q-val-list
                                       additional-parameter-list))
         (b-val-list (zl:lexpr-funcall #'extract-b-val-list
                                       additional-parameter-list))
         (orientation-list (zl:lexpr-funcall #'extract-orientation-list
                                             additional-parameter-list))
         (number-of-processing-layers
          (- num-of-layers 1))
         (num-of-s-layers (/ number-of-processing-layers 2)))
; error checking and layer creation
    (cond ((not (= num-of-s-layers (length r-val-list)
                   (length q-val-list) (length b-val-list)))
                                                  ; check extracted parameters
           (ferror "Improper parameters for a net with ~D S-layers:
                    r value list = ~s,
                    q value list = ~s,
                    b value list = ~s."
                   num-of-s-layers r-val-list q-val-list b-val-list))
          ((not (= num-of-layers
                   (length planes-per-layer-list)
                   (length plane-size-list)))       ; check passed parameters
           (ferror "Improper parameters for a net with ~D layers:
                    Either not enough planes sizes listed in ~s, OR
                     not enough plane sizes listed in ~s."
                   num-of-layers planes-per-layer-list
                   plane-size-list))
          ((not (= number-of-processing-layers (length mask-size-list)))
                                                  ; check projection masks
           (ferror "Improper parameters for a net with ~D layers beyond
                    input layer:
                      Not enough connection mask sizes listed in ~s."
                   number-of-processing-layers mask-size-list))
; Create appropriate number of layers, one at a time, and record as a list.
; For each layer, extract appropriate parameter settings and sizes.
          (t
           (do* ((i 1 (+ i 1))
                 (r-val-list r-val-list (cdr r-val-list))
                 (r-val (car r-val-list) (car r-val-list))
                 (q-val-list q-val-list (cdr q-val-list))
                 (q-val (car q-val-list) (car q-val-list))
                 (b-val-list b-val-list (cdr b-val-list))
                 (b-val (car b-val-list) (car b-val-list))
                 (rest-orientations orientation-list
                                    (cddr rest-orientations))
                 (s-orientations (car rest-orientations)
                                 (car rest-orientations))
                 (c-orientations (cadr rest-orientations)
                                 (cadr rest-orientations))
                 (prev-plane-num (car planes-per-layer-list)
                                 (cadr planes-per-layer))
                 (planes-per-layer (cdr planes-per-layer-list)
                                   (cddr planes-per-layer))
                 (num-of-s-planes (car planes-per-layer)
                                  (car planes-per-layer))
                 (mask-list mask-size-list (cddr mask-list))
                 (mask-size (car mask-list) (car mask-list))
                 (plane-sizes-list (cdr plane-size-list)
                                   (cddr plane-sizes-list))
                 (prev-c-plane-size (car plane-size-list)
                                    c-plane-size)
                 (s-plane-size (car plane-sizes-list)
                               (car plane-sizes-list))
                 (c-plane-size (cadr plane-sizes-list)
                               (cadr plane-sizes-list))
; create input layer
                 (input-layer
                  (make-layer :planes
```

(continued on page 95)

Listing Two *(Listing continued, text begins on page 28.)*

```
                              (create-plane-list
                               1 prev-c-plane-size
                               (do ((i 1 (+ i 1))
                                    (times (apply #'* prev-c-plane-size))
                                    (res '((0)) (cons '(0) res)))
                                   ((>= i times) res))
                               0 0 1 mask-size 'C)
                              :type 'C))
; create connections
                  (s-connection-list
                   (create-connection-list 1 prev-c-plane-size s-plane-size
                                           con-pattern mask-size 'S)
                                  ; Connections same for for all planes
                   (create-connection-list 1 prev-c-plane-size s-plane-size
                                           con-pattern mask-size 'S))
                  (c-connection-list
                   (create-connection-list 1 s-plane-size c-plane-size
                                           con-pattern (cadr mask-list) 'C)
                                         ;C-cells connect one S-plane
                   (create-connection-list 1 s-plane-size c-plane-size
                                           con-pattern (cadr mask-list) 'C))
; create planes
    (s-planes
      (create-plane-list num-of-s-planes s-plane-size s-connection-list
                         prev-plane-num b-val s-orientations mask-size 'S)
      (create-plane-list num-of-s-planes s-plane-size s-connection-list
                         prev-plane-num b-val s-orientations mask-size 'S))
    (c-planes
     (create-plane-list (cadr planes-per-layer) c-plane-size c-connection-list
                        num-of-s-planes b-val c-orientations mask-size 'C)
     (create-plane-list (cadr planes-per-layer) c-plane-size c-connection-list
                        num-of-s-planes b-val c-orientations mask-size 'C))
;assign layers
    (new-s-layer (make-layer :planes s-planes :e-connections s-connection-list
                :local-parameters '(:net-type neocognitron :r ,r-val :q ,q-val)
                                  :prev-layer input-layer :type 'S)
                 (make-layer :planes s-planes :e-connections s-connection-list
                :local-parameters '(:net-type neocognitron :r ,r-val :q ,q-val)
                                   :prev-layer (car (last layers))
                                   :type 'S))
    (new-c-layer (make-layer :planes c-planes :e-connections c-connection-list
                 :local-parameters '(:net-type neocognitron :r ,r-val :q ,q-val)
                                   :prev-layer new-s-layer :type 'C)
                 (make-layer :planes c-planes :e-connections c-connection-list
                 :local-parameters '(:net-type neocognitron :r ,r-val :q ,q-val)
                                   :prev-layer new-s-layer :type 'C))
; add new layers to layer list
                  (layers               ; S and C layers
                   (list input-layer new-s-layer new-c-layer)
                   (append layers
                           (list new-s-layer new-c-layer))))
                  ((>= i num-of-s-layers) layers)))   ; return list of layers
          )
    )
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;                Keyword and Parameter EXTRACTION functions

(defun EXTRACT-TYPE-KEY (&key net-type &allow-other-keys)
  net-type)                             ;; Extract the NET-TYPE keyed value
(defun EXTRACT-R-VAL-LIST (&key r-val-list &allow-other-keys)
  r-val-list)                           ;; Extract the R-VAL-LIST keyed value
(defun EXTRACT-Q-VAL-LIST (&key q-val-list &allow-other-keys)
  q-val-list)                           ;; Extract the Q-VAL-LIST keyed value
(defun EXTRACT-B-VAL-LIST (&key b-val-list &allow-other-keys)
  b-val-list)                           ;; Extract the B-VAL-LIST keyed value
(defun EXTRACT-ORIENTATION-LIST (&key orientation-list &allow-other-keys)
  orientation-list)                  ;; Extract the ORIENTATION-LIST keyed value
(defun EXTRACT-NET-  (net)          ;; Extract the   parameter from a NET
  (zl:lexpr-funcall #'extract-net- -aux
                    (net-local-parameters net)))
(defun EXTRACT-NET- -AUX (&key   &allow-other-keys)
   )                                ;; Extract the   keyed value
(defun EXTRACT-LAYER-R (layer)      ;; Extract the R parameter from a LAYER
  (zl:lexpr-funcall #'extract-layer-r-aux
                    (layer-local-parameters layer)))
(defun EXTRACT-LAYER-R-AUX (&key r &allow-other-keys)
  r)                                ;; Extract the R keyed value
(defun EXTRACT-LAYER-Q (layer)      ;; Extract the Q parameter from a LAYER
  (zl:lexpr-funcall #'extract-layer-q-aux
                    (layer-local-parameters layer)))
(defun EXTRACT-LAYER-Q-AUX (&key q &allow-other-keys)
  q)                                ;; Extract the Q keyed value

(defun EXTRACT-PLANE-B (plane)      ;; Extract the B parameter from a PLANE
  (zl:lexpr-funcall #'extract-plane-b-aux
                    (plane-local-parameters plane)))
(defun EXTRACT-PLANE-B-AUX (&key b &allow-other-keys)
  b)                                ;; Extract the B keyed value

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;                Keyword and Parameter INSERTION functions

(defun INSERT-KEYED-VAL (old-list insertion-list)  ;; Insert a keyed value
  (do ((list (cddr old-list) (cddr list))
       (i-key (car insertion-list))
       (o-key (car old-list) (car list))
       (o-val (cadr old-list) (cadr list))
       (result insertion-list                    ; insert new keyed value
             (append result                            ; keep old values
                  (if (not (equal o-key i-key))        ;  for other keys
                      (list o-key o-val)))))
      ((null o-key) result)))
```

End Listing Two

(continued on page 96)

Listing Three *(Listing continued, text begins on page 28.)*

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: andy; Base: 10 -*-

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;       A Research Environment for the Instantiation of Neural Networks
;;;       Andrew  J. Czuchry, Jr. -- Georgia Institute of Technology
;;;       Georgia Tech Research Institute -- Artificial Intelligence Branch
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;                          Net TRAINING functions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Top level function for training.  Trains on all patterns in PATTERN-LIST.
;;   Sets mappings.
;     Trains on *neocognitron* by default.
(defun TRAIN-MAIN-LOOP (&key
          (net *neocognitron*)
          (pattern-list (mapcar #'item-misc (scl:send lrn-pattrns :item-list)))
          (iteration-list '(4 4 4 4))
          print-data)
(let* ((net-type (x):lexpr-funcall &"extract-type-key
                                        (net-locate-parameters net)))
       (iterations (compute-iteration-list net-type i iteration list)
       (actual-its (    iterations 1)))
  (do* ((i 0 (+ i 1))
      (conditioned-it-list  (select-iteration-list net-type i iteration-list)))
      (terminate-iterations net-type i iterations))
  (do* ((patterns pattern-list (cdr patterns))       ; loop over all patterns
        (pattern (car patterns) (car patterns))
        (actual-its (length pattern-list)))
       (cond (print-data
              (format t "~%~%")
              (time:print-current-time)
              (format t "~%% ** training ~s on pattern ~s.
                  (Iteration ~d of ~d, ~d more patterns.)"
                      net pattern i actual-its (length (cdr patterns)))))
       (train pattern net iteration-list)                ; perform training
       )
}
  (create-mappings net pattern-list)      ; record mapping between pattern and
                                          ;  most active "output layer" cell
  (cond (print-data
         (format t "~%~%")
         (time:print-current-time)))
  )
}

;; Trains a net to recognize a pattern. Returns the plane/cell of the final
;  layer which responds most actively to the pattern.
(defun TRAIN (pattern-plane net
              &optional (iterations-per-layer-list '(4 4 4 1)))
  (let* ((layers-to-be-trained (cdr (net-layers net)))
         (num-layers (length layers-to-be-trained)))
    (cond ((not (= num-layers (length iterations-per-layer-list)))
           (ferror "~% Improper training iteration count list for
               training ~D layers: ~s"
                  num-layers iterations-per-layer-list))
          (t
           (setf (plane-cells (car (layer-planes (car (net-layers net)))))
                 pattern-plane)    ; assign input pattern
           (do* ((layer-list (cdr (net-layers net)) (cdr layer-list))
                                    ; loop over all layers
                 (layer (car layer-list) (car layer-list))
                 (iteration-list iterations-per-layer-list
                                 (cdr iteration-list))
                 (iterations (car iteration-list) (car iteration-list))
                                    ; # times to train layer
                 (result (caar (train-layer layer iterations))
                                    ; train each layer in succession
                         (caar (train-layer layer iterations))))
;                (result (caar (update-layer layer))    ; update the layer
;                        (caar (update-layer layer))))
                ((null (cdr layer-list)) result))
                                    ; return most active cell in final layer
           )
          )
    )
  )

; Trains a layer in a net to recognize a pattern. Continues training until
;  updating layer produces no more changes in the representative list
;  (returned by UPDATE-LAYER). At some point I'd like to remove the
;  ITERATIONS parameter and work only from changes in rep list, but
;  it is computationally prohibitive in the current version of the system.

(defun TRAIN-LAYER (layer &optional iterations)
  (do* ((old-reps nil new-reps)   ; record most active cell
        (new-reps (update-layer layer) (update-layer layer))
                                  ; re-adjust after training
        (i 0 (+ i 1)))
    ((or (equal old-reps new-reps) (>= i iterations)) new-reps)
                                  ; check if training complete
    (train-layer-aux layer new-reps))     ; perform training
  )

; Trains a layer in a net to recognize a pattern
(defun TRAIN-LAYER-AUX (layer &optional (representative-data-list
   (update-layer layer)))
    (let ((net-type (zl:lexpr-funcall #'extract-type-key
                                      (layer-local-parameters layer))))
; select appropriate training routine
      (zl:selectq  net-type
        (neocognitron
                (train-neocognitron-layer-aux
                  layer representative-data-list))
        (DIANN
                (train-DIANN-layer-aux
```

(continued on page 98)

Listing Three *(Listing continued, text begins on page 28.)*

```
                layer representative-data-list))
        (ART2
                (train-ART2-layer-aux
                  layer representative-data-list))
        (backpropagation
                (train-backprop-layer-aux
                  layer representative-data-list))
                )
      )
    )

; Trains a layer in a neocognitron to recognize a pattern
(defun TRAIN-NEOCOGNITRON-LAYER-AUX (layer &optional (representative-data-list
 (update-layer layer)))
   (cond ((> *trace* 3)
          (format t "~%training layer.~%  Chosen Representative list:~%  ~s"
                  representative-data-list)))
  (cond ((eq (layer-type layer) 'S)                    ; only train S-layers
         (mapcar
           #'(lambda (representative-data)
               (let* ((plane (car representative-data))
                      (pos (cadadr representative-data))
                      (prev-layer (layer-prev-layer layer))
                      (q-val (extract-layer-q layer))
                      (results
                        (do* ((all-connections (layer-i-connections layer)
                                               (cdr all-connections))
                              (connections (connections-for-pos layer plane pos))
                                                   ; extract connections
                              (all-i-vals-list (plane-i-weights plane)
                                               (cdr all-i-vals-list))
                                                ; extract current weights
                              (old-i-vals (car all-i-vals-list)
                                          (car all-i-vals-list))
                              (all-i-vals-list (plane-i-weights plane)
                                               (cdr all-i-vals-list))
                              (all-i-vals (car all-i-vals-list)
                                          (car all-i-vals-list))
                              (all-i-weights-list (plane-i-weights plane)
                                                  (cdr all-i-weights-list))
                              (all-i-weights (car all-i-weights-list)
                                             (car all-i-weights-list))
                              (raw-result (train-plane prev-layer
                                                       connections old-i-vals
                                          all-i-vals all-i-weights q-val)
                                  ; perform training on excitatory connections
                                (train-plane prev-layer connections old-i-vals
                                             all-i-vals all-i-weights q-val))
                              (result raw-result
                                      (list (append (car result)
                                                    (car raw-result))
                                            (+ (cadr result)
                                               (cadr raw-result)))))
                                                       ; record result
                             ((null (cdr all-connections)) result)))
                                                       ; return result
                        (new-i-weight-list (car results))
                                                  ; adjust inhibitory weights
                 (new-b-val (* q-val (compute-inhib-input (connections-for-pos layer plane pos)
                                                          (c-weights-for-pos  plane pos)
                                                          prev-layer))))
                 (setf (plane-i-weights plane)
                       (transpose-on-type new-i-weight-list (layer-type layer)))
                 (setf (plane-local-parameters plane)
                  (insert-keyed-val (plane-local-parameters plane) '(:b ,new-b-val)))))
                   representative-data-list))
                ((eq (layer-type layer) 'C)   representative-data-list)
                                    ; for C-layers, return representatives
                )
         )

;Trains a plane's CONNECTIONS to all planes in previous layer
(defun TRAIN-PLANE (prev-layer connections old-i-val-lists all-i-val-lists
                                                        all-i-weight-lists q-val)
 (do* ((connection-list connections (cdr connection-list))
                                                          ; extract connections
       (connection (car connection-list) (car connection-list))
       (c-val-list all-i-val-lists (cdr c-val-list)) ; extract current weights
       (c-vals (car c-val-list) (car c-val-list))
       (c-weight-list all-i-weight-lists (cdr c-weight-list))
       (c-weights (car c-weight-list) (car c-weight-list))
       (rev-old-i-vals (reverse old-i-val-lists))
       (old-i-vals (car old-i-val-lists)
                   (nth (- (length c-val-list) 1) rev-old-i-vals))
       (con-vals (train-plane-aux prev-layer connection old-i-vals
                                  c-vals c-weights q-val)
                 (train-plane-aux prev-layer connection old-i-vals
                                  c-vals c-weights q-val))    ; perform actual training
       (new-i-weights (list (car con-vals))
                      (nconc new-i-weights
                             (list (car con-vals))))
       (vtotal (cadr con-vals) (+ vtotal (cadr con-vals))))
      ((null (cdr connection-list)) (list (list new-i-weights) vtotal)))
                                          ; return new connection weights
 )
```

End Listing Three

Listing Four

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: andy; Base: 10 -*-

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;       A Research Environment for the Instantiation of Neural Networks
;;;       Andrew  J. Czuchry, Jr. -- Georgia Institute of Technology
;;;       Georgia Tech Research Institute -- Artificial Intelligence Branch
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;                        IDENTIFICATION functions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Attempts to recognize a pattern using NET as a trained net
;  Returns the plane of the final layer which responds to the pattern.
(defun IDENTIFY (pattern-plane net)
  (setf (plane-cells (car (layer-planes (car (net-layers net)))))
        pattern-plane)                                    ; assign input pattern
  (let ((result (caar (last (update-net net)))))          ; update net
    (if (listp result)
        (car result))           ; return most active cell of final layer
    ))

; Updates entire net
;  Returns maximum value as nested set of lists
;   (((plane (value, pos)) ... (plane (value, pos)) layer1)
;   (((plane (value, pos)) ... (plane (value, pos)) layer2) ...)
(defun UPDATE-NET (net)
  (let ((  (extract-net-  net)))
    (do* ((layer-list  (cdr (net-layers net)) (cdr layer-list))
                                                        ; loop over all layers
          (layer (car layer-list) (car layer-list))
          (layer-max (update-layer layer) (update-layer layer))
                                                          ; update the layer
          (max (list (append layer-max (list layer)))
                                ; max value ((value, pos) plane)... layer) list
               (append max (list (append layer-max (list layer))))))
                                                          ; append each layer
         ((null (cdr layer-list)) max)
      )
    )
  )
```

End Listing Four

Listing Five

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: andy; Base: 10 -*-

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;       A Research Environment for the Instantiation of Neural Networks
;;;       Andrew  J. Czuchry, Jr. -- Georgia Institute of Technology
;;;       Georgia Tech Research Institute -- Artificial Intelligence Branch
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;           Sample Variables of networks to be instantiated
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defvar *neocognitron-net*
        '(create-net
           :num-of-layers 7
           :num-of-planes-per-layer-list '(1 24 24 24 24 24 24)
           :plane-size-list '((16 16) (16 16) (10 10) (8 8) (6 6) (2 2) (1 1))
           :connection-pattern 'square
           :mask-size-list '((5 5) (5 5) (5 5) (5 5) (5 5) (2 2))
           :net-parameters '(:  0.5)
           :additional-parameter-list '(:net-type neocognitron
                                        :r-val-list (4.0 1.5 1.5)
                                        :q-val-list (1.0 16.0 16.0)
                                        :b-val-list (0.0 0.0 0.0)
                                        :orientation-list (8 1 8 1))
           )
  )
(defvar *neocognitron-net2*
        '(create-net
           :num-of-layers 7
           :num-of-planes-per-layer-list '(1 15 20 20 24 20 10)
           :plane-size-list '((24 24) (18 18) (12 12) (9 9) (8 8) (4 4) (1 1))
           :connection-pattern 'square
           :mask-size-list '((7 7) (3 3) (3 3) (4 4) (5 5) (4 4))
           :net-parameters '(:  0.5)
           :additional-parameter-list '(:net-type neocognitron
                                        :r-val-list (3.0 1.0 1.0)
                                        :q-val-list (10.0 18.0 18.0)
                                        :b-val-list (0.0 0.0 0.0)
                                        :orientation-list (12 1 12 1))
           )
        )
```

End Listings

Listing One *(Text begins on page 46.)*

```
/*************************************************************************\
 *  tswit.c  -- iRMX II task switch time measurement.
 *  Compiler: iC-286 V4.1 (LARGE model). Q1 1989, by R. P. Kar
\*************************************************************************/

#include <stdio.h>
#include <rmxc.h>

#define MAX_LOOPS 500000L

unsigned       el_time, pri, status;
unsigned long  strt_sec, end_sec;
selector       task1_t, task2_t;
unsigned long  count1, count2;
float          ts_time;

/* "union" used to decompose a pointer into segment:offset */
typedef struct {unsigned offset; selector sel;} ptr_s;
union { unsigned *pointer; ptr_s ptr; } ptr_u;

void task1()
{
  for (count1 = 0; count1 < MAX_LOOPS; count1++)
    rqsleep(0, &status);                    /* Task switch happens here */
  rqdeletetask(NULL, &status);                          /* delete self */
}

void task2()
{
  for (count2 = 0; count2 < MAX_LOOPS; count2++)
    rqsleep(0, &status);                    /* Task switch happens here */
  rqdeletetask(NULL, &status);                          /* delete self */
}

/************************  MAIN PROGRAM  ************************/

main()
{

printf("\nTask Switch measurement\n   Each task runs %D times...\n\n",
          MAX_LOOPS);

/* Measure execution time of task1 and task2 when they are executed
   serially (without task switching). */
strt_sec = rqgettime(&status);                  /* Start of timing period */

  for (count1 = 0; count1 < MAX_LOOPS; count1++)
    /* rqsleep(0, &status) */ ;
  for (count2 = 0; count2 < MAX_LOOPS; count2++)
    /* rqsleep(0, &status) */ ;
end_sec = rqgettime(&status);                    /* End of timing period */

el_time = (unsigned)(end_sec - strt_sec);

/* Place a pointer to any variable in union "ptr_u", so the data segment
   of this program becomes known. */
ptr_u.pointer = &status;

/* Get main program's priority level */
pri = rqgetpriority (NULL, &status);

/* Create two (identical) tasks, which just switch between themselves */
task1_t = rqcreatetask (pri+1, task1, ptr_u.ptr.sel, 0L, 512, 0, &status);
if (status != 0) printf("rqcreatetask error\n");

task2_t = rqcreatetask (pri+1, task2, ptr_u.ptr.sel, 0L, 512, 0, &status);
strt_sec = rqgettime(&status);                  /* Start of timing period */

/* Set main program's priority below task 1,2 so they run to completion */
rqsetpriority( (selector)0, pri+2, &status );
rqsleep( 0, &status );

end_sec = rqgettime(&status);                    /* End of timing period */

/* Set main program back to initial priority */
rqsetpriority( (selector)0, pri, &status );

el_time = (unsigned)(end_sec - strt_sec) - el_time;
ts_time = ( (float)el_time * 1000000.0 ) / ((float)MAX_LOOPS * 2.0) ;
printf("   Task switch time = %5.1f microseconds\n", ts_time);
dqexit(0);
}
```

End Listing One

Listing Two

```
/*************************************************************************\
 *  preempt.c  -- iRMX II preemption time benchmark.
 *  Measures the time for 1 preemptive task switch + 1 non-preemptive task
 *  switch. Compiler: iC-286 V4.1 (LARGE model). Q4 1989, by R. P. Kar
\*************************************************************************/

#include <stdio.h>
#include <rmxc.h>

/* NOTE: 100,000 iterations takes about 35 minutes on a 16 MHz 386 PC */
#define MAX_LOOPS 100000L

/* Note: This is a CPU-dependent value. It must be set such that
 * the execution time for this loop: for (j=0; j < ONE_TICK; j++) spare++
 * is slightly longer than one iRMX sleep period.
*/
#define ONE_TICK 4200

unsigned       pri, status, i, spare, el_time;
unsigned long  strt_sec, end_sec;
selector       task1_t, task2_t, co_conn;
unsigned long  count1, count2;
float          preempt_time;

/* "union" used to decompose a pointer into segment:offset */
typedef struct {unsigned offset; selector sel;} ptr_s;
union { unsigned *pointer; ptr_s ptr; } ptr_u;

/* The lower priority task. It sits in delay loop waiting to be preempted. */
void task1()
{
  unsigned loc_status;
  for (count1 = 0; count1 < MAX_LOOPS; count1++)
    for (i = 0; i < ONE_TICK; i++) ++spare;             /* Waste time */
  printf("deleting task 1\n\n");
  rqdeletetask(NULL, &loc_status);                      /* delete self */
}

/* The higher priority task. When it goes to sleep (once in every loop) iRMX
 * makes a non-preemptive switch to the other task; when the sleep period ends
 * this task preempts the other task.
 */
void task2()
{
  unsigned loc_status;
  for (count2 = 0; count2 < MAX_LOOPS; count2++)
    /* When rqsleep is called, task switch to lower priority task happens.
     * When 1 clock period is over, other task is preempted and control
     * returns to the next line.
     */
    rqsleep(1, &loc_status);
  printf("\ndeleting task 2\n");
  rqdeletetask(NULL, &loc_status);                /* delete self */
}

/************************  MAIN PROGRAM  ************************/
main()
{

printf("\nPreemption time benchmark\n   Each task runs %D times...\n\n",
          MAX_LOOPS);

/* Measure execution time of task1 and task2 when they are executed
 * serially (without task switching or preemption).
 */
strt_sec = rqgettime(&status);                  /* Start of timing period */
  for (count1 = 0; count1 < MAX_LOOPS; count1++)
    for (i = 0; i < ONE_TICK; i++) ++spare;
  for (count2 = 0; count2 < MAX_LOOPS; count2++)
end_sec = rqgettime(&status);                    /* End of timing period */

el_time = (unsigned)(end_sec - strt_sec);

printf("    Execution without premption & task switching took %u seconds\n",
              el_time);

/* Place a pointer to any variable in union "ptr_u", so the data segment
   of this program becomes known.
 */
ptr_u.pointer = &status;

/* Get main program's priority */
pri = rqgetpriority (NULL, &status);

task1_t = rqcreatetask (pri+2, task1, ptr_u.ptr.sel, 0L, 512, 0, &status);
if (status != 0) printf("rqcreatetask error\n");

task2_t = rqcreatetask (pri+1, task2, ptr_u.ptr.sel, 0L, 512, 0, &status);

strt_sec = rqgettime(&status);                  /* Start of timing period */

/* Set main program's priority below task 1,2 so they run to completion */
rqsetpriority( (selector)0, pri+3, &status );
rqsleep( 0, &status );

end_sec = rqgettime(&status);                  /* End of timing period */

/* Set main program back to initial priority */
rqsetpriority( (selector)0, pri, &status );
el_time = (unsigned)(end_sec - strt_sec) - el_time;
preempt_time = ( (float)el_time / (float)MAX_LOOPS ) * 1000000.0;
printf("     Preemption time + task switch time = %5.1f microseconds\n",
              preempt_time);
dqexit(0);
}
```

End Listing Two

Listing Three

```
/*************************************************************************\
 *  ltncy.c -- iRMX II interrupt latency benchmarking program.
 *  Method:  This program first sets up an interrupt handler for an unused
 *    interrupt level. It then reads the count in the system timer (timer
 *    0 on the 8254 chip) and simulates an external interrupt to the CPU by
 *    a cause$interrupt instruction. The interrupt handler latches timer 0,
 *    so this program can read it again after the handler returns control.
 *    The difference in the two timer-count values is the interrupt latency.
 *                  Oct 1989, by R. P. Kar
\*************************************************************************/

#include <stdio.h>
#include <rmxc.h>

/* Define base address of 8254 (Programmable Interval Timer) chip */
#define        PIT_ADDR   0x40

unsigned       status, ticks, timer_cnt1, timer_cnt2;
unsigned       dummy_w;
unsigned char  pri, lo_cnt1, hi_cnt1, lo_cnt2;

extern void int_hndlr();

/************************  MAIN PROGRAM  ************************/
main ()
```

```
{
printf("                    ***  WARNING ***\n\n");
printf("  This program assumes that timer and interrupt controller\n");
printf("  hardware is fully compatible with the IBM PC/AT\n\n");

/* Set up local handler for IRQ3 on master 8259 */
rqsetinterrupt( 0x38, 0, int_hndlr, (selector)0, &status );
disable();                                          /* Disable interrupts */

/* Latch and read timer 0 value. Interrupt handler will latch it again */
outbyte( PIT_ADDR + 3, 0 );

/* The following two instructions read the value latched in counter 0. They
    are unavoidable measurement overhead and inflate the interrupt latency
    by a few clock cycles.
 */
lo_cnt1 = inbyte( PIT_ADDR );
hi_cnt1 = inbyte( PIT_ADDR );

/* Activate the interrupt handler. It will latch timer 0 and return. */
causeinterrupt(59);

/* The interrupt handler has latched the timer 0 count. Now read it. */
lo_cnt2 = inbyte( PIT_ADDR );
dummy_w = (unsigned int)inbyte( PIT_ADDR );
timer_cnt2 = (unsigned int)lo_cnt2 + (dummy_w << 8);

enable();                                           /* Re-enable interrupts */

dummy_w = (unsigned int)hi_cnt1;
timer_cnt1 = (unsigned int)lo_cnt1 + (dummy_w << 8);

/* Calculate difference in timer counts (timer counts DOWN to 0) */
if (timer_cnt1 > timer_cnt2)
    ticks = timer_cnt1 - timer_cnt2;
else                            /* Rare case when timer has wrapped around */
     ticks = timer_cnt1 + (0xffff - timer_cnt2 + 1);

/* Display results */
printf("   Interrupt latency = %u timer ticks\n", ticks);

/* Note that timer is pulsed by 1.19 MHz crystal */
printf("                     = %4.1f microseconds\n\n", ((float)ticks)/1.19 );

rqresetinterrupt( 0x38, &status );
dqexit(0);
}
```

End Listing Three

Listing Four

```
; latch.asm -- Interrupt handler. Merely latches timer 0 in a
; PC/AT (or hardware compatible computer).

      NAME latch

latch SEGMENT PUBLIC

int_hndlr PROC FAR
PUBLIC int_hndlr
  PUSHA
  XOR  AX,AX
  OUT  43H, AL    ; Latch 8254 counter 0
  POPA
  IRET
int_hndlr ENDP
latch ENDS
      END
```

End Listing Four

Listing Five

```
/*****************************************************************************\
 *  semshuf.c -- iRMX II semaphore shuffle measurement.
 *  Measures the latency (within iRMX) for a task to acquire
 *  a sempahore that is owned by another equal-priority task.
 *  Compiler: Intel iC-286 V4.1 (LARGE model). Q3 1989, by R. P. Kar
\*****************************************************************************/

#include <stdio.h>
#include <rmxc.h>

#define MAX_LOOPS 100000L

enum     YESNO {NO, YES} sem_exch;
unsigned       el_time, status;
selector       task1_t, task2_t, sem_t;
unsigned char  pri;
unsigned long  count1, count2, maxloop2;
unsigned long  strt_sec, end_sec;
float          semshuf_time;

/* "union" used to decompose a pointer into segment:offset */
typedef struct {unsigned offset; selector sel;} ptr_s;
union { unsigned *pointer; ptr_s ptr; } ptr_u;

void task1()
{
  unsigned rem_units, t1_status;
  for (count1 = 0; count1 < MAX_LOOPS; count1++)
  {
    /* Task waits here until other task relinquishes semaphore */
    if (sem_exch == YES)
      rem_units = rqreceiveunits( sem_t, 1, 0xffff, &t1_status );
    rqsleep(0, &t1_status);
    if (sem_exch == YES)
      rqsendunits( sem_t, 1, &t1_status );
    rqsleep(0, &t1_status);
```

(continued on page 102)

Listing Five *(Listing continued, text begins on page 46.)*

```
  }
  rqdeletetask( (selector)0, &t1_status );                    /* delete self */
}

void task2()
{
  unsigned rem_units, t2_status;
  for (count2 = 0; count2 < MAX_LOOPS; count2++)
  {
    /* Task waits here until other task relinquishes semaphore */
    if (sem_exch == YES)
      rem_units = rqreceiveunits( sem_t, 1, 0xffff, &t2_status);
    rqsleep(0, &t2_status);
    if (sem_exch == YES)
      rqsendunits( sem_t, 1, &t2_status );
    rqsleep(0, &t2_status);
  }
  rqdeletetask( (selector)0, &t2_status );                    /* delete self */
}

/************************  MAIN PROGRAM  ************************/
main()
{

printf("\nSemaphore shuffle benchmark\n   %U shuffles...\n\n", MAX_LOOPS*2);

/* Get priority of main program */
pri = rqgetpriority( (selector)0, &status );

/* Create 2 tasks; measure their execution time WITHOUT semaphore shuffling */
 sem_exch = NO;

task1_t = rqcreatetask( pri+1, task1, ptr_u.ptr.sel, 0L, 512, 0, &status );
if (status != 0) printf("Create task error\n");

task2_t = rqcreatetask( pri+1, task2, ptr_u.ptr.sel, 0L, 512, 0, &status );

strt_sec = rqgettime(&status);                  /* Start of timing period */

/* Set main program's priority below task 1,2 so they run to completion */
rqsetpriority( (selector)0, pri+2, &status );
rqsleep( 0, &status );

end_sec = rqgettime(&status);                   /* End of timing period */

el_time = (unsigned)(end_sec - strt_sec);
printf("    Execution time without semaphore shuffle = %u secs\n",el_time);

/* Set main() back to original priority level */
rqsetpriority( (selector)0, pri, &status );

sem_t = rqcreatesemaphore( 1, 1, 0, &status );
if (status != 0) printf("Create sem error\n");

/* Re-create 2 tasks. This time they will shuffle semaphore between them. */
sem_exch = YES;

task1_t = rqcreatetask( pri+1, task1, ptr_u.ptr.sel, 0L, 512, 0, &status );
if (status != 0).printf("Create task error\n");

task2_t = rqcreatetask( pri+1, task2, ptr_u.ptr.sel, 0L, 512, 0, &status );

strt_sec = rqgettime(&status);                    /* Start of timing period */

/* Set main program's priority below task 1,2 so they run to completion */
rqsetpriority( (selector)0, pri+2, &status);
rqsleep( 0, &status );

end_sec = rqgettime(&status);                    /* End of timing period */
el_time = (unsigned)(end_sec - strt_sec) - el_time;
printf("    %U semaphore exchanges took %u seconds\n", MAX_LOOPS*2, el_time)

semshuf_time = ( (float)el_time / ((float)MAX_LOOPS * 2.0) ) * 1000000.0;
printf("    ..... %5.1f microseconds per shuffle\n\n", semshuf_time);

dqexit(0);
}
```

End Listing Five

Listing Six

```
/*************************************************************************\
 *  deadbrk.c -- iRMX II Deadlock break-time measurement.
 *  A low, medium and high priority task is created. Deadlock occurs
 *  when the following chronological sequence happens:
 *     (1) low priority task takes exclusive control of a critical resource
 *     (2) medium or high priority task preempts it.
 *     (3) high priority task requests resource; gets suspended
 *     (4) Medium priority task runs, blocking other two tasks indefinitely
 *  This situation is handled in iRMX by acquiring a "region" before
 *  using critical resource, and relinquishing it after use. This benchmark
 *  measures the overhead involved in "breaking the deadlock".
 *  Compiler: Intel iC-286 V4.1 (LARGE model). Q3 1989, by R. P. Kar
\*************************************************************************/

#include <stdio.h>
#include <rmxc.h>

#define MAX_LOOPS 10000
/* Note: This is a CPU-dependent value. It must be set such that the
 * execution time for this loop: for (j=0; j < DELAY; j++) spare++
 * is slightly longer than one iRMX sleep period.
 */
#define DELAY     4000

unsigned       el_time, spare, status;
selector       task1_t, task2_t, task3_t, region_t;
unsigned char  pri;
enum           YESNO {NO, YES} dead_brk;
```

```
unsigned long  count1, count2, count3, max_loops;
unsigned long  strt_sec, end_sec;
float          deadbrk_time;

/* "union" used to decompose a pointer into segment:offset */
typedef struct {unsigned offset; selector sel;} ptr_s;
union { unsigned *pointer; ptr_s ptr; } ptr_u;

/* Low priority task */
void task1()
{
  unsigned t1_status, j;
  while (1)
  {
    if (count1 == max_loops)
    { printf("deleting task1\n");
      rqdeletetask( (selector)0, &t1_status );           /* delete self */
    }
    /* Get control over critical region */
    rqreceivecontrol( region_t, &t1_status );

    for (j = 0; j < DELAY; j++) spare++;                 /* delay loop */

    count1++;
    rqsendcontrol( &t1_status );
  }
}

/* Medium priority task. Only uses CPU time and sleep periodically. */
void task2()
{
  unsigned j, t2_status;

  while (1)
  {
    if (count2 == max_loops)
    { printf("deleting task2\n");
      rqdeletetask( (selector)0, &t2_status );          /* delete self */
    }
    for (j = 0; j < DELAY/4; j++) spare++;               /* delay loop */

    rqsleep(1, &t2_status);
    count2++;
  }
}

/* High priority task. Potential deadlock when it tries to gain control
   of the "region" resource, because low-priority task holds region mostly.
 */
void task3()
{
  unsigned t3_status;

  while (1)
  {
    if (count3 == max_loops)
    { printf("deleting task3\n");
      rqdeletetask( (selector)0, &t3_status );          /* delete self */
    }
    rqsleep(1, &t3_status);
    /* Ask for control of the region. Relinquish control immediately after
       receiving it. If task1 is not already holding region, this should
       take very little time. Otherwise, OS must break deadlock.
     */
    if (dead_brk == YES)
    { rqreceivecontrol( region_t, &t3_status );
      rqsendcontrol( &t3_status );
    }

    count3++;
  }
}

/********************** Main program ***********************/
main( argc, argv )

unsigned argc;
char *argv[];

{
if (argc > 1)
  max_loops = (unsigned)atoi(argv[1]);
else max_loops = MAX_LOOPS;

printf("\nDeadlock break time benchmark\n   %U loops...\n\n",max_loops);

/* Get priority of main program */
pri = rqgetpriority( (selector)0, &status );

/* Create three tasks. Task1 has lowest priority, task3 has highest.
 * Measure their execution time WITHOUT deadlocks.
 */
count1 = count2 = count3 = 0;
dead_brk = NO;

task1_t = rqcreatetask( pri+3, task1, ptr_u.ptr.sel, 0L, 512, 0, &status );
if (status != 0) printf("Create task error\n");

task2_t = rqcreatetask( pri+2, task2, ptr_u.ptr.sel, 0L, 512, 0, &status );

task3_t = rqcreatetask( pri+1, task3, ptr_u.ptr.sel, 0L, 512, 0, &status );

strt_sec = rqgettime(&status);                  /* Start of timing period */

/* Set main program's priority below task 1,2,3 so they run to completion */
rqsetpriority( (selector)0, pri+4, &status );
while ( (count1 < max_loops) || (count2 < max_loops) || (count3 < max_loops) )
  rqsleep( 10, &status );
```

(continued on page 104)

Listing Six *(Listing continued, text begins on page 46.)*

```
end_sec = rqgettime(&status);                    /* End of timing period */

el_time = (unsigned)(end_sec - strt_sec);
printf("    Execution time without deadlocks = %u secs\n\n",el_time);

/* Set main() back to original priority level */
rqsetpriority( (selector)0, pri, &status );

/* Create a "region". To ensure mutually exclusive access to a critical
   resource a task must acquire the region first */
region_t = rqcreateregion( 1, &status );
if (status != 0) printf("Create region error\n");

count1 = count2 = count3 = 0;
dead_brk = YES;

/* Re-create tasks 1,2,3. Now tasks 1 & 3 will compete for region */
task1_t = rqcreatetask( pri+3, task1, ptr_u.ptr.sel, 0L, 512, 0, &status );
if (status != 0) printf("Create task error\n");

task2_t = rqcreatetask( pri+2, task2, ptr_u.ptr.sel, 0L, 512, 0, &status );

task3_t = rqcreatetask( pri+1, task3, ptr_u.ptr.sel, 0L, 512, 0, &status );

strt_sec = rqgettime(&status);                   /* Start of timing period */

/* Set main program's priority below tasks 1,2,3 so they run to completion */
rqsetpriority( (selector)0, pri+4, &status);
while ( (count1 < max_loops) || (count2 < max_loops) || (count3 < max_loops) )
  rqsleep( 10, &status );

end_sec = rqgettime(&status);                    /* End of timing period */

el_time = (unsigned)(end_sec - strt_sec) - el_time;
printf("    %U deadlock resolutions took %u seconds\n", count3, el_time);

deadbrk_time = ( (float)el_time/(float)count3 ) * 1000000.0;
printf("    ..... %6.1f microseconds per resolution\n\n\n", deadbrk_time);

dqexit(0);
}
```

End Listing Six

Listing Seven

```
/*****************************************************************\
 *  it_msg.c -- iRMX II inter-task data message latency measurement.
 *  First run the code of two tasks serially (no messages sent). Then
 *  create two tasks and a "mailbox" and measure how much extra time is
 *  needed to send a fixed number of messages from task 1 to task 2.
 *      Compiler: iC-286 V4.1 (LARGE model). Q4 1989, by R. P. Kar
\*****************************************************************/

#include <stdio.h>
#include <rmxc.h>

#define MAX_LOOPS 200000L

unsigned long  strt_sec, end_sec;
selector       task1_t, task2_t, mbox_t;
unsigned       pri, el_time, msg_length, status;
unsigned long  count1, count2;
float          it_msg_time;
char           msg_buf[10] = "MESSAGE\0",
               recv_buf[];

/* "union" used to decompose a pointer into segment:offset */
typedef struct {unsigned offset; selector sel;} ptr_s;
union { unsigned *pointer; ptr_s ptr; } ptr_u;

/* This task sends data messages, to task 2 that is waiting to receive */
void task1()
{
  unsigned loc_status;

  for (count1 = 0; count1 < MAX_LOOPS; count1++)
  {  /* Put a serial # on the message */
     msg_buf[8] = (unsigned char)count1 / 256;
     rqsenddata( mbox_t, msg_buf, 10, &loc_status );
  }
  printf("Task 1 exiting....\n");
  rqdeletetask(NULL, &status);                          /* delete self */
}

/* This task receives the data messages */
void task2()
{
  unsigned loc_status;

  for (count2 = 0; count2 < MAX_LOOPS; count2++)
    msg_length = rqreceivedata( mbox_t, recv_buf, 0xffff, &loc_status );
  printf("    Last message received... %s %u (length %u)\n", recv_buf,
                          (unsigned)recv_buf[8], msg_length );
  rqdeletetask(NULL, &status);                          /* delete self */
}

/**************************  MAIN PROGRAM  *************************/
/* First parameter to "rqcreatemailbox" ==> data mailbox, FIFO queues */
#define MBOX_FLAG 0x0020

main()
{
printf("   Inter-task message latency measurement\n");
printf("      Sending %D data messages...\n\n", MAX_LOOPS);

/* Set up a mailbox for inter-task data communication */
mbox_t = rqcreatemailbox( MBOX_FLAG, &status );
if (status != 0) printf("rqcreatemailbox error\n");

/* Measure serial execution time of tasks 1,2 (without messages) */

strt_sec = rqgettime(&status);                   /* Start of timing period */
  for (count1 = 0; count1 < MAX_LOOPS; count1++)
  {  /* Put a serial # on the message */
     msg_buf[8] = (unsigned char)count1 / 256;
     /* rqsenddata( mbox_t, msg_buf, 10, &loc_status ); */
  }
  for (count2 = 0; count2 < MAX_LOOPS; count2++)
    /* msg_length = rqreceivedata( mbox_t, recv_buf, 0xffff, &loc_status ) */;
end_sec = rqgettime(&status);                    /* End of timing period */

el_time = (unsigned)(end_sec - strt_sec);

/* Place a pointer to any variable in union "ptr_u", so the data segment
   of this program becomes known.
 */
ptr_u.pointer = &status;

/* Get main program's priority level */
pri = rqgetpriority (NULL, &status);

task1_t = rqcreatetask (pri+2, task1, ptr_u.ptr.sel, 0L, 512, 0, &status);
if (status != 0) printf("rqcreatetask error\n");

/* Task 2 is created with a higher priority than task 1. This ensures that if
 * it is waiting at a mailbox for a message from task 1, it will be scheduled
 * as soon as the message is sent.
 */
task2_t = rqcreatetask (pri+1, task2, ptr_u.ptr.sel, 0L, 512, 0, &status);

strt_sec = rqgettime(&status);                   /* Start of timing period */

/* Set main program's priority below task 1,2 so they run to completion */
rqsetpriority( (selector)0, pri+3, &status );
rqsleep( 0, &status );

end_sec = rqgettime(&status);                    /* End of timing period */

/* Set main program back to initial priority */
rqsetpriority( (selector)0, pri, &status );

el_time = (unsigned)(end_sec - strt_sec) - el_time;
it_msg_time =  ( (float)el_time * 1000000.0 ) / (float)MAX_LOOPS ;
printf("    Inter-task message latency + task switch time = %6.1f
microsecs\n",
         it_msg_time);

/* Delete mailbox */
rqdeletemailbox( mbox_t, &status );

dqexit(0);
}
```

End Listings

Listing One *(Text begins on page 56.)*

```
/* FONT1 STRUCTURE DEFINITIONS */

struct font1head {      /* standard character font header */
   unsigned char fnttype;       /* font structure type: */
        /* non-compressed type = 0x16 or compressed type = 0x14 */
   char fntname[13];   /* font name: always followed with a '.set' extension*/
   unsigned char fntcheck; /* check digit: verifies a Data Transforms font: */
        /* non-compressed font = 0xba, compressed font = 0xdc */
   unsigned char fntbase;
                /* baseline count (in pixels) from top to bottom, top = 0 */
   unsigned char fnttotal; /* total characters in font:*/
        /* limited to the lower 94 ASCII characters: 0x21 - 0x7E */
   unsigned char fntstart;      /* starting character */
   unsigned char fntstatus; /* proportional or non-proportional: 0=non-prop.*/
   unsigned char fnthsize;     /* horizontal cell size in pixels */
   unsigned char fntvsize;     /* vertical cell size in pixels */
   unsigned char fntbytes;     /* number of horizontal bytes in current cell*/
   unsigned char fntspaceh;    /* space bar horizontal size in pixels */
   unsigned char fntchargap;   /* pixels between characters default */
   unsigned char fntlfgap;     /* pixels between linefeeds default */
   int fntlength;              /* total length of file */
   unsigned char fntpitch;     /* italics pitch (0 = none) */
              /* bits 0 - 6 ... number of scanlines to skip */
              /* bit  7 ... 0 = decrement xpos, 1 = increment xpos */
   unsigned char fntinvert;     /* 0 = dont invert, 1 = invert */
   unsigned char fnthbold;      /* number of overlapping bits horizontal */
   unsigned char fntvbold;      /* number of overlapping bits vertical */
   unsigned char fnthmag;       /* integral horizontal bit magnification */
   unsigned char fntvmag;       /* integral vertical bit magnification */
   unsigned char fnthfract;     /* fractional horizontal bit magnification */
   unsigned char fntvfract;     /* fractional vertical bit magnification */
   unsigned char fntdirection;
                    /* Print direction 0=left to right,1..3=counterclock */
   unsigned char fntrot90;      /* rotation 0=up, 1..3=counterclock 1...3 */
   unsigned char fnthflip;      /* horizontal flip 0 = no, 1 = yes */
   unsigned char fntvflip;      /* vertical flip 0 = no, 1 = yes */
   unsigned char fntcolor;      /* color of font */
             /* bits 0 - 3 ... foreground color */
             /* bit 0 ... strike black ribbon */
             /* bit 1 ... strike blue ribbon */
             /* bit 2 ... strike red ribbon */
             /* bit 3 ... strike yellow ribbon */
             /* bits 4 - 7 ... background color */
             /* bit 4 ... strike black ribbon */
             /* bit 5 ... strike blue ribbon */
             /* bit 6 ... strike red ribbon */
             /* bit 7 ... strike yellow ribbon */
   unsigned char fntsubtype;    /* subcategory type of this font */
             /* 0 = normal font subtype */
             /* 1 = equation roman font subtype */
             /* 2 = equation symbol font subtype */
   unsigned char fntunused[18]; /* unused bytes */
};

struct font1 {  /* standard character font (type = 0x16) */
   struct font1head fhd;        /* font header */
   char *fntcellptr[FONT1TOTAL + 1];
                   /* offsets from beginning of file to character bitmaps */
   char fntcellwidth[FONT1TOTAL + 1];   /* cell widths if proportional */
};
```

End Listing One

Listing Two

```
/* FONT2 STRUCTURE DEFINITIONS */

struct font2 {
   struct font1head fhd2;                    /* font header */
   unsigned int fnt2cellseg[FONT2TOTAL + 1];
                                  /* array: segment pointers to characters*/
   int fnt2cellhsize[FONT2TOTAL + 1];   /* array: cell horiz sizes in bits */
   int fnt2cellhoffset[FONT2TOTAL + 1]; /* array: cell horiz offsets in bits*/
   int fnt2cellhbytes[FONT2TOTAL + 1];  /* array: cell horiz size in bytes */
   int fnt2cellvsize[FONT2TOTAL + 1];   /* array: cell vert sizes in bits */
   int fnt2cellvoffset[FONT2TOTAL + 1]; /* array: cell vert offsets in bits */
};
```

End Listings

Listing One *(Text begins on page 72.)*

```
/*************************************************************************
File:    Kabooooom.c
Purpose: Allows the user to 'walk' through a minefield; a detector shows
  how many mines are immediately adjacent to you. As you visit a cell, it
  leaves a marker telling you how many were next to you, and you have the
  ability to mark cells with a character (assumably to mark mines).
 Changes:
 11/10/89 (tdeii) If you call the program with "/s" or "/S", it gives
  you a "safer" game, where it does not let you walk on spaces that you
  have marked (whether there is a mine there or not!)
  11/11/89 (tdeii) Allows you to press "?" and get some help starting at your
  current position; will mark mines that it knows (by deducing their position),
  and "visit" places that it knows are safe. This propagates until it cannot
  deduce anything else (see EvaluatePosition).
*************************************************************************/
#include "stdio.h"
#include "stdarg.h"
#include "stdlib.h"
#include "dos.h"
#include "conio.h"
#include "string.h"

#define SCREEN_X 80
#define SCREEN_Y 25
#define GRID_X 15
#define GRID_Y  9
#define TRUE 1
#define FALSE 0

#define bEMPTY    0
#define bVISITED  1
#define bBOMB     2
#define bCURRENT  3
#define bFINISH   4
#define bEXPLODED 5

#define MAKECOLOR(fore,back) ((back)*16+(fore))

typedef int BOOL;
typedef struct tagADJACENCYGROUP {
  int BombCount;          /* Number of bombs located in this adj. group */
  int CellCount;          /* Number of cells filled                     */
  int Cell[8][2];         /* x,y coordinates of up to 8 cells           */
} ADJACENCYGROUP;

int  Board[GRID_X][GRID_Y];          /* Board; see codes above (bXXX)     */
int  UserMark[GRID_X][GRID_Y];       /* User marks; 0 = none, 'M' = mine  */
int  nNumMines;                      /* Number of mines on board          */
int  UserX, UserY;                   /* Current user X and Y position     */
BOOL bShowBombs;                     /* TRUE if program shows bombs (it   */
                                     /* does this after you win or lose)  */
BOOL bSafeGame;                      /* TRUE if program does not let you  */
                                     /* walk on mines you have marked     */
ADJACENCYGROUP AdjacencyGroup[GRID_X][GRID_Y]; /* AG for each board pos   */
char szClear[79] = "
                   ";
void Pause(void)
{
  if (getch() == 0) getch();
}

void GetXY_(int *pX, int *pY)
{
  union REGS regs;
  regs.h.ah = 3;
  regs.h.bh = 0;                                        /* display page 0 */
  int86(0x10, &regs, &regs);
  if (pY != NULL) {
    (*pY) = regs.h.dh;
  }
  if (pX != NULL) {
    (*pX) = regs.h.dl;
  }
}
void GotoXY_(int x, int y)
{
  union REGS regs;
  regs.h.ah = 2;
  regs.h.dh = (unsigned char)y;
  regs.h.dl = (unsigned char)x;
  regs.h.bh = 0;                                        /* display page 0 */
  int86(0x10, &regs, &regs);
}
int nRandom(int nMax)
{
  return ((int)((double)rand() / RAND_MAX * (double)nMax));
}
void DisplayChar(int x, int y, char cChar, int nColor)
{
  char far *cPos;
  if ((x>=0 && x<SCREEN_X) &&
      (y>=0 && y<SCREEN_Y)) {
    cPos = MK_FP( 0x0b800,((x + y*80) << 1));
    *cPos = cChar;
    *(cPos+1) = (char)nColor;
  }
}
int CountMines(int x, int y)
{
  int i, j;
  int nCount;
  nCount = 0;
  for (i=-1; i<=1; i++) {
    for (j=-1; j<=1; j++) {
      if ((x+i >= 0) && (x+i < GRID_X) &&
          (y+j >= 0) && (y+j < GRID_Y)) {
        if ((Board[x+i][y+j] == bBOMB) ||
            (Board[x+i][y+j] == bEXPLODED)) {
          nCount++;
        }
      }
    }
  }
  return (nCount);
}
void DisplayCell(int x, int y)
{
  int Char;
  Char = UserMark[x][y];
  if (Char == 0) {
    Char = 32;
  }
  DisplayChar(x*4+1, y*2+1, Char, MAKECOLOR(14,1));
  DisplayChar(x*4+3, y*2+1, Char, MAKECOLOR(14,1));
  switch (Board[x][y]) {
    case bEMPTY:                                        /** Empty cell   **/
      Char = ' ';
      break;
    case bVISITED:                                      /** Visited cell **/
      Char = '0' + CountMines(x, y);
      break;
    case bBOMB:                                         /** Bomb cell!   **/
      if (bShowBombs) {
        Char = 15;
      } else {
        Char =' ';
      }
      break;
    case bCURRENT:                                      /** Current pos  **/
      Char = 2;
      break;
    case bFINISH:                                       /** Finish cell  **/
      Char = 19;
      break;
    case bEXPLODED:                                     /** Exploded!    **/
      Char = 15;
      break;
  }
  if (Char != 0) {
    DisplayChar(x*4+2, y*2+1, Char, MAKECOLOR(14,1));
  }
}
void Initialize(void)
{
  unsigned int     nRand;       /* seed for random number generator        */
  struct dostime_t sDosTime;    /* time structure; used for above seed     */
  _dos_gettime(&sDosTime);
  nRand = (unsigned int)((sDosTime.hsecond * 600) +
                         (sDosTime.second * 10) +
                         (sDosTime.minute / 6));
  srand(nRand);
}
void PaintBoard(void)
{
  int x, y, i;
  for (x=0; x<SCREEN_X; x++) {
    for (y=0; y<SCREEN_Y; y++) {
      DisplayChar(x, y, ' ', MAKECOLOR(14, 1));
    }
  }
  /** Draw left and right sides **/
  DisplayChar(0, 0, 218, MAKECOLOR(14,1));  /*upper left corner */
  DisplayChar(GRID_X*4, 0, 191, MAKECOLOR(14,1)); /*upper right corner */
  for (y=1; y<=GRID_Y; y++) {
    DisplayChar(0, y*2, 195, MAKECOLOR(14,1));  /* left edge */
    DisplayChar(GRID_X*4, y*2, 180, MAKECOLOR(14,1)); /* right edge */
  }
  DisplayChar(0, GRID_Y*2, 192, MAKECOLOR(14,1));  /* lower left corner */
  DisplayChar(GRID_X*4, GRID_Y*2, 217, MAKECOLOR(14,1)); /*lower right */
  /** Draw inside corners **/
  for (x=1; x<GRID_X; x++) {
    DisplayChar(x*4, 0, 194, MAKECOLOR(14,1)); /* top edge */
    for (y=1; y<GRID_Y; y++) {
      DisplayChar(x*4, y*2, 197, MAKECOLOR(14,1)); /* intersections */
    }
    DisplayChar(x*4, GRID_Y*2, 193, MAKECOLOR(14,1)); /* bottom edge */
  }
  /** Draw connecting lines **/
  for (x=0; x<=GRID_X; x++) {
    for (y=0; y<=GRID_Y; y++) {
      if (y != GRID_Y) {
        DisplayChar(x*4, y*2+1, 179, MAKECOLOR(14,1)); /* verticals  */
      }
      if (x != GRID_X) {
        for (i=1; i<4; i++) {
          DisplayChar(x*4+i, y*2, 196 , MAKECOLOR(14,1)); /* horizontals */
        }
      }
    }
  }
  GotoXY_(0, SCREEN_Y - 1);
  for (x=0; x<GRID_X; x++) {
    for (y=0; y<GRID_Y; y++) {
      DisplayCell(x, y);
    }
  }
}
void SetUpBoard(void)
{
  int i, j;
  int nMines;
  BOOL bDone;
  char cBuffer[80];
  bShowBombs = FALSE;
  /** First, get number of bombs **/
  nNumMines = 0;
  while ((nNumMines < 10) || (nNumMines > 40)) {
    GotoXY_(0, 24);
```

(continued on page 110)

Listing One *(Listing continued, text begins on page 72.)*

```
    printf("How many bombs do you want? (10-40)?? ");
    fgets(cBuffer, sizeof(cBuffer), stdin);
    sscanf(cBuffer, "%d", &nNumMines);
  }
  /** next, clear out board & user scratchpad **/
  for (i=0; i<GRID_X; i++) {
    for (j=0; j<GRID_Y; j++) {
      Board[i][j] = 0;
      UserMark[i][j] = 0;
    }
  }
  for (nMines=0; nMines<nNumMines; nMines++) {
    bDone = FALSE;
    while (!bDone) {
      i = nRandom(GRID_X);          /* First you roll it, */
      j = nRandom(GRID_Y);          /* Then you pat it,   */
      if ((Board[i][j] == bEMPTY) &&
          (!((i <= 1) && (j <= 1))) &&
          (!((i >= GRID_X - 2) && (j >= GRID_Y - 2)))
         ) {
        bDone = TRUE;
      }
    }
    Board[i][j] = bBOMB;            /* Then you mark it with a 'B' */
  }
  /* Set user at position 0, 0 */
  UserX = 0;
  UserY = 0;
  Board[0][0] = bCURRENT;
  /* Set finish (hq) at position GRID_X, GRID_Y */
  Board[GRID_X - 1][GRID_Y - 1] = bFINISH;
  /* Display board on screen */
  PaintBoard();
}
BOOL Travel(int dx, int dy)
{
  int NewX, NewY;             /* New X and Y coordinates of user       */
  BOOL bInvalid;              /* TRUE if trying to walk off board      */
  BOOL bAbort;                /* TRUE if user won or lost (abort game) */
  BOOL bBombWalk;             /* TRUE if user tried to walk on a bomb  */

  bAbort = FALSE;
  NewX = UserX + dx;
  NewY = UserY + dy;
  bInvalid = FALSE;
  bBombWalk = FALSE;
  if ((NewX < 0) || (NewX >= GRID_X)) {
    bInvalid = TRUE;
  }
  if ((NewY < 0) || (NewY >= GRID_Y)) {
    bInvalid = TRUE;
  }
  if ((!bInvalid) && (bSafeGame) && (UserMark[NewX][NewY] == 'M')) {
    bInvalid = TRUE;
    bBombWalk = TRUE;
  }
  if (bInvalid) {
    GotoXY_(0, SCREEN_Y - 1);
    printf("** INVALID MOVE ** ... press any key...");
    if (bBombWalk) {
      printf("(You must un-mark it.)");
    }
    Pause();
    GotoXY_(0, SCREEN_Y - 1);
    printf(szClear);
  } else {
    if (Board[NewX][NewY] == bBOMB) {
      bAbort = TRUE;
      Board[UserX][UserY] = bVISITED;
      DisplayCell(UserX, UserY);
      Board[NewX][NewY] = bEXPLODED;
      DisplayCell(NewX, NewY);
      GotoXY_(0, 22);
      printf("******** YOU HAVE STEPPED ON A BOMB!! ********");
      Pause();
      GotoXY_(0, 22);
      printf(szClear);
      GotoXY_(0, 22);
    } else {
      if ((NewX == GRID_X-1) && (NewY == GRID_Y-1)) {
        bAbort = TRUE;
        Board[UserX][UserY] = bVISITED;
        DisplayCell(UserX, UserY);
        Board[NewX][NewY] = bCURRENT;
        DisplayCell(NewX, NewY);
        GotoXY_(0, 22);
        printf("************* YOU HAVE WON!! *************");
        Pause();
        GotoXY_(0, 22);
        printf(szClear);
        GotoXY_(0, 22);
      } else {
        Board[UserX][UserY] = bVISITED;
        DisplayCell(UserX, UserY);
        UserX = NewX;
        UserY = NewY;
        Board[UserX][UserY] = bCURRENT;
        DisplayCell(UserX, UserY);
      }
    }
  }
  GotoXY_(0, GRID_Y*2+2);
  printf("Number of mines around you: %d", CountMines(UserX, UserY));
  GotoXY_(0, SCREEN_Y - 1);
  return (bAbort);
}
void PlaceUserMark(void)
{
  BOOL bDone, bAbort;
  int  Ch;
  int  NewX, NewY;
  int  dx, dy;

  bAbort = FALSE;
  GotoXY_(0, 24);
  printf("Mark in which direction? (ESC=abort)");
  bDone = FALSE;
  while (!bDone) {
    bDone = TRUE;
    Ch = getch();
    switch (Ch) {
      case 0:
        Ch = getch();
        switch (Ch) {
          case 71:  /* home */
            dx = -1;
            dy = -1;
            break;
          case 72:  /* up arrow */
            dx = 0;
            dy = -1;
            break;
          case 73:  /* page up */
            dx = 1;
            dy = -1;
            break;
          case 75:  /* left arrow */
            dx = -1;
            dy = 0;
            break;
          case 77:  /* right arrow */
            dx = 1;
            dy = 0;
            break;
          case 79:  /* end */
            dx = -1;
            dy = 1;
            break;
          case 80:  /* down arrow */
            dx = 0;
            dy = 1;
            break;
          case 81:  /* page down */
            dx = 1;
            dy = 1;
            break;
          default:
            bDone = FALSE;
            break;
        }
        break;
      case '7':  /* home */
        dx = -1;
        dy = -1;
        break;
      case '8':  /* up arrow */
        dx = 0;
        dy = -1;
        break;
      case '9':  /* page up */
        dx = 1;
        dy = -1;
        break;
      case '4':  /* left arrow */
        dx = -1;
        dy = 0;
        break;
      case '6':  /* right arrow */
        dx = 1;
        dy = 0;
        break;
      case '1':  /* end */
        dx = -1;
        dy = 1;
        break;
      case '2':  /* down arrow */
        dx = 0;
        dy = 1;
        break;
      case '3':  /* page down */
        dx = 1;
        dy = 1;
        break;
      case 27:
      case 13:
      case 10:
      case  8:
        bAbort = TRUE;
        break;
      default:
        bDone = FALSE;
        break;
    }
  }
  GotoXY_(0, 24);
  printf(szClear);
  if (!bAbort) {
    NewX = UserX + dx;
    NewY = UserY + dy;
    if ((NewX < 0) || (NewX >= GRID_X) || (NewY < 0) || (NewY >= GRID_Y)) {
      GotoXY_(0, 24);
      printf("ERROR: Out of bounds!!");
      Pause();
      GotoXY_(0, 24);
      printf(szClear);
    } else {
      GotoXY_(0, 24);
      if (UserMark[NewX][NewY] != 0) {
        Ch = 0;
      } else {
```

```
        Ch = 'M';
      }
      UserMark[NewX][NewY] = Ch;
      DisplayCell(NewX, NewY);
    }
  }
  GotoXY_(0, 24);
}
void ComputeAdjacency(int x, int y)
{
  int dX, dY;
  int BombCount;
  int Cell;
  if ((x >= 0) && (x < GRID_X) && (y >= 0) && (y < GRID_Y)) {
    if ((Board[x][y] == bVISITED) || (Board[x][y] == bCURRENT)) {
      BombCount = CountMines(x, y);
      Cell = 0;
      for (dX=-1; dX<=1; dX++) {
        for (dY=-1; dY<=1; dY++) {
          if (!((dX == 0) && (dY == 0))) {
            if ((x+dX >= 0) && (x+dX < GRID_X) &&
                (y+dY >= 0) && (y+dY < GRID_Y)) {
              if ((Board[x+dX][y+dY] != bVISITED) &&
                  (Board[x+dX][y+dY] != bCURRENT)) {
                if (UserMark[x+dX][y+dY] != 0) {
                  BombCount--;
                } else {
                  AdjacencyGroup[x][y].Cell[Cell][0] = x+dX;
                  AdjacencyGroup[x][y].Cell[Cell][1] = y+dY;
                  Cell++;
                }
              }
            }
          }
        }
      }
      AdjacencyGroup[x][y].BombCount = BombCount;
      AdjacencyGroup[x][y].CellCount = Cell;
    } else {
      AdjacencyGroup[x][y].CellCount = 0;
      AdjacencyGroup[x][y].BombCount = -1;   /** Don't look flag */
    }
  }
}
int AddToPositionList(int PositionList[GRID_X * GRID_Y][2],
                      int PositionListHead, int x, int y)
{
  int  nIndex;
  BOOL bFound;
  ComputeAdjacency(x, y);
  bFound = FALSE;
  for (nIndex=0; (nIndex<PositionListHead) && (!bFound); nIndex++) {
    if ((PositionList[nIndex][0] == x) && (PositionList[nIndex][1] == y)) {
      bFound = TRUE;
    }
  }
  if (!bFound) {
    PositionList[PositionListHead][0] = x;
    PositionList[PositionListHead][1] = y;
    PositionListHead++;
  }
  if (PositionListHead > GRID_X * GRID_Y) {
    GotoXY_(0, 22);
    printf("ERROR! PositionListHead > max (%d)", PositionListHead);
    Pause();
    GotoXY_(0, 22);
    printf(szClear);
    GotoXY_(0, 22);
  }
  return (PositionListHead);
}
int AddSurroundingToPositionList(int PositionList[GRID_X * GRID_Y][2],
                                 int PositionListHead, int x, int y)
{
  int dX, dY;

  for (dX=-1; dX<=1; dX++) {
    for (dY=-1; dY<=1; dY++) {
      if ((x+dX >= 0) && (x+dX < GRID_X) && (y+dY >= 0) && (y+dY < GRID_Y)) {
        if ((Board[x+dX][y+dY] == bVISITED) ||
            (Board[x+dX][y+dY] == bCURRENT)) {
          PositionListHead = AddToPositionList(PositionList, PositionListHead,
x+dX, y+dY);
        }
      }
    }
  }
  return (PositionListHead);
}
BOOL FindPositionInAG(ADJACENCYGROUP *pAG, int x, int y)
{
  int  nIndex;
  BOOL bFound;
  bFound = FALSE;
  for (nIndex=0; nIndex<pAG->CellCount; nIndex++) {
    if ((pAG->Cell[nIndex][0] == x) && (pAG->Cell[nIndex][1] == y)) {
      bFound = TRUE;
    }
  }
  return (bFound);
}
void MarkBombCell(int x, int y)
{
  UserMark[x][y] = 'M';
  DisplayCell(x, y);
  if (Board[x][y] != bBOMB) {
    GotoXY_(0, 22);
    printf("LOGIC ERROR: I tagged a phantom bomb @ (%d,%d).", x, y);
    Pause();
```

(continued on page 112)

Listing One *(Listing continued, text begins on page 72.)*

```
    GotoXY_(0, 22);
    printf(szClear);
    GotoXY_(0, 24);
  }
}
void VisitCell(int x, int y)
{
  if (Board[x][y] != bCURRENT) {
    if (Board[x][y] == bBOMB) {
      GotoXY_(0, 22);
      printf("LOGIC ERROR: I walked on a bomb @ (%d,%d).", x, y);
      Pause();
      GotoXY_(0, 22);
      printf(szClear);
      GotoXY_(0, 24);
    }
    Board[x][y] = bVISITED;
    DisplayCell(x, y);
  }
}
int CountCommonCells(ADJACENCYGROUP *pGroup1, ADJACENCYGROUP *pGroup2)
{
  int Cell, nCount;
  nCount = 0;
  for (Cell=0; Cell<pGroup1->CellCount; Cell++) {
    if (FindPositionInAG(pGroup2,
                         pGroup1->Cell[Cell][0], pGroup1->Cell[Cell][1])) {
      nCount++;
    }
  }
  return (nCount);
}
BOOL ProcessRule3(ADJACENCYGROUP *pCurrentAG, ADJACENCYGROUP *pTempAG,
                  int PositionList[GRID_X * GRID_Y][2],
                  int *pPositionListHead)
{
  int x;
  int BombCount, CellCount;
  int PositionListHead;
  int CellHolder[9][2];
  int CellHolderHead;
  BOOL bRetVal;
  PositionListHead = *pPositionListHead;
  bRetVal = FALSE;
  BombCount = pCurrentAG->BombCount;
  CellCount = pCurrentAG->CellCount;
  if (pTempAG->CellCount == CountCommonCells(pTempAG, pCurrentAG)) {
    BombCount -= pTempAG->BombCount;
    CellCount -= pTempAG->CellCount;
    if ((CellCount > 0) && ((BombCount == CellCount) || (BombCount == 0))) {
      bRetVal = TRUE;
      CellHolderHead = 0;
      CellCount = pCurrentAG->CellCount;
      for (x=0; x<CellCount; x++) {
        if (!FindPositionInAG(pTempAG, pCurrentAG->Cell[x][0],
                              pCurrentAG->Cell[x][1])) {
          if (BombCount == 0) {
            VisitCell(pCurrentAG->Cell[x][0], pCurrentAG->Cell[x][1]);
          } else {
            MarkBombCell(pCurrentAG->Cell[x][0], pCurrentAG->Cell[x][1]);
          }
          /* Queue up cells to put in position list for later */
          CellHolder[CellHolderHead][0] = pCurrentAG->Cell[x][0];
          CellHolder[CellHolderHead][1] = pCurrentAG->Cell[x][1];
          CellHolderHead++;
        }
      }
      for (x=0; x<CellHolderHead; x++) {
        PositionListHead = AddSurroundingToPositionList(
                           PositionList,
                           PositionListHead,
                           CellHolder[x][0],
                           CellHolder[x][1]);
      }
    }
  }
  *pPositionListHead = PositionListHead;
  return (bRetVal);
}
void EvaluatePosition(void)
{
  int  CurrentX, CurrentY;
  int  x, y;
  int  Cell;
  int  dX, dY;
  int  BombCount, CellCount;
  int  PositionList[GRID_X * GRID_Y][2], PositionListHead;
  ADJACENCYGROUP *pTempAG;
  BOOL bDone;
  BOOL bModifiedAny;
  bModifiedAny = TRUE;
  for (x=0; x<GRID_X; x++) {
    for (y=0; y<GRID_Y; y++) {
      ComputeAdjacency(x, y);
    }
  }
  PositionList[0][0] = UserX;
  PositionList[0][1] = UserY;
  PositionListHead = 1;
  while (bModifiedAny) {
    bModifiedAny = FALSE;
    while (PositionListHead > 0) {
      CurrentX = PositionList[0][0];
      CurrentY = PositionList[0][1];
      for (x=0; x<PositionListHead-1; x++) {
        PositionList[x][0] = PositionList[x+1][0];
        PositionList[x][1] = PositionList[x+1][1];
      }
      PositionListHead--;
      ComputeAdjacency(CurrentX, CurrentY);
      BombCount = AdjacencyGroup[CurrentX][CurrentY].BombCount;
      CellCount = AdjacencyGroup[CurrentX][CurrentY].CellCount;
      if ((CellCount > 0) && (BombCount > -1)) {
/*
        Rule 1: if number of bombs = number of cells, all are bombs!
*/
        if (CellCount == BombCount) {
          for (Cell=0; Cell<CellCount; Cell++) {
            x = AdjacencyGroup[CurrentX][CurrentY].Cell[Cell][0];
            y = AdjacencyGroup[CurrentX][CurrentY].Cell[Cell][1];
            MarkBombCell(x, y);
            PositionListHead = AddSurroundingToPositionList(PositionList,
                                                            PositionListHead,
                                                            x, y);
            bModifiedAny = TRUE;
          }
        } else {
/*
          Rule 2: if number of bombs = 0, all cells are ok!
*/
          if ((BombCount == 0) && (CellCount > 0)) {
            for (Cell=0; Cell<CellCount; Cell++) {
              x = AdjacencyGroup[CurrentX][CurrentY].Cell[Cell][0];
              y = AdjacencyGroup[CurrentX][CurrentY].Cell[Cell][1];
              VisitCell(x, y);
              PositionListHead = AddToPositionList(PositionList,
                                                   PositionListHead,
                                                   x, y);
              PositionListHead = AddSurroundingToPositionList(PositionList,
                                                              PositionListHead,
                                                              x, y);
              bModifiedAny = TRUE;
            }
          } else {
/*
            Rule 3: if AG completely overlaps another AG, subtract 2nd
                    # of bombs from 1st; check rules 1 & 2. If rule 1 or
                    2 is true in this case, stop looking in rule 3.
*/
            bDone = FALSE;
            for (Cell=0; (Cell<CellCount) && (!bDone); Cell++) {
              x = AdjacencyGroup[CurrentX][CurrentY].Cell[Cell][0];
              y = AdjacencyGroup[CurrentX][CurrentY].Cell[Cell][1];
              for (dX=-1; (dX<=1) && (!bDone); dX++) {
                for (dY=-1; (dY<=1) && (!bDone); dY++) {
                  if ((x+dX >= 0) && (x+dX < GRID_X) &&
                      (y+dY >= 0) && (y+dY < GRID_Y)) {
                    pTempAG = &AdjacencyGroup[x+dX][y+dY];
                    if (pTempAG->BombCount > 0) {       /* if == 0, no help! */
                      bDone = ProcessRule3(&AdjacencyGroup[CurrentX][CurrentY],
                                           pTempAG,
                                           PositionList,
                                           &PositionListHead);
                      if (bDone) {
                        bModifiedAny = TRUE;
                      }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
    if (bModifiedAny) {
      for (x=0; x<GRID_X; x++) {
        for (y=0; y<GRID_Y; y++) {
          if ((Board[x][y] == bVISITED) || (Board[x][y] == bCURRENT)) {
            PositionListHead = AddToPositionList(PositionList,
                                                 PositionListHead,
                                                 x, y);
          }
        }
      }
    }
  }
}
BOOL LetUserMove(void)
{
  BOOL bDone;
  BOOL bQuit;
  int  Ch;
  bDone = FALSE;
  while (!bDone) {
    Ch = getch();
    switch (Ch) {
      case 0:
        Ch = getch();
        switch (Ch) {
          case 71:  /* home */
            bDone = Travel(-1, -1);
            break;
          case 72:  /* up arrow */
            bDone = Travel(0, -1);
            break;
          case 73:  /* page up */
            bDone = Travel(1, -1);
            break;
          case 75:  /* left arrow */
            bDone = Travel(-1, 0);
            break;
          case 77:  /* right arrow */
            bDone = Travel(1, 0);
            break;
          case 79:  /* end */
            bDone = Travel(-1, 1);
            break;
          case 80:  /* down arrow */
```

```
            bDone = Travel(0, 1);
            break;
          case 81:  /* page down */
            bDone = Travel(1, 1);
            break;
        }
        break;
      case '7':  /* home */
        bDone = Travel(-1, -1);
        break;
      case '8':  /* up arrow */
        bDone = Travel(0, -1);
        break;
      case '9':  /* page up */
        bDone = Travel(1, -1);
        break;
      case '4':  /* left arrow */
        bDone = Travel(-1, 0);
        break;
      case '6':  /* right arrow */
        bDone = Travel(1, 0);
        break;
      case '1':  /* end */
        bDone = Travel(-1, 1);
        break;
      case '2':  /* down arrow */
        bDone = Travel(0, 1);
        break;
      case '3':  /* page down */
        bDone = Travel(1, 1);
        break;
      case 'Q':
      case 'q':
      case 27:
        bDone = TRUE;
        break;
      case 'M':
      case 'm':
        PlaceUserMark();
        break;
      case '?':
        EvaluatePosition();
        break;
    }
  }
  bShowBombs = TRUE;
  PaintBoard();
  GotoXY (0, SCREEN_Y - 1);
  printf("Again (Y/n)? ");
  bDone = FALSE;
  while (!bDone) {
    Ch = getch();
    if ((Ch == 'Y') || (Ch == 'y') || (Ch == 13) || (Ch == 10)) {
      bDone = TRUE;
      bQuit = FALSE;
      printf("Y\n");
    }
    if ((Ch == 'N') || (Ch == 'n')) {
      bDone = TRUE;
      bQuit = TRUE;
      printf("N\n");
    }
    if (Ch == NULL) {
      getch();
    }
  }
  return (bQuit);
}
int main(int argc, char *argv[])
{
  BOOL bDone;
  Initialize();
  if ((argc > 1) &&
      (argv[1][0] == '/') &&
      ((argv[1][1] == 's') || (argv[1][1] == 'S'))) {
    bSafeGame = TRUE;
    printf("SAFE GAME in effect.\n");
  } else {
    bSafeGame = FALSE;
  }
  bDone = FALSE;
  while (!bDone) {
    SetUpBoard();
    bDone = LetUserMove();
  }
  return (0);
}
```

End Listing

Listing One *(Text begins on page 77.)*

```
/* eesc.c -- edge enhancement system in C */
#include <stdio.h>
#define ASize(x)        (sizeof(x)/sizeof(x[0]))        /* length of array */

/* PrintGraph() - print out graph of an array of numbers*/
FILE    *PFOutFp  = (stdout);
int PrintGraph( PFarray, ILen, Iny )
float           *PFarray;       /* pointer to floating piont array */
int              ILen;          /* length of the array */
int              Iny;           /* # of points along the y-axis */
{
    float        FMin, FMax;    /* minimum and maximum values */
    float        FSc, FOff;     /* scale & offset */
    int          Iwx;           /* work index */
    int          Ilx;           /* line index */
    int          ITx;           /* temp index */
    int          IpTx;          /* prior line index */
    int          Ich;           /* character to display */
    /* --- check that all parameters are "reasonable" --- */
    if ( PFarray == (float *)0 || ILen <= 0 || Iny <= 1 )
        return( -1 );
    /* --- compute minimum and maximum values for array --- */
    FMin = PFarray[0];
    FMax = PFarray[0];
    for( Iwx = 1; Iwx < ILen; Iwx++ ) {
        if ( FMin > PFarray[Iwx] )      FMin = PFarray[Iwx];
        if ( PFarray[Iwx] > FMax )      FMax = PFarray[Iwx];
    }
    if ( FMin > 0.0 ) FMin = 0.0;
    /* --- from minimum and maximum, compute scale and offset --- */
    if ( (FMax - FMin) < .0001 ) {
        /* --- assume that all values are the same --- */
        FSc  = 1.0;
        FOff = -FMin;
    } else {
        FSc  = Iny / (FMax - FMin);
        FOff = -FSc * FMin;
    }
    IpTx = 0;
    fputc( '\n', PFOutFp );
    for( Ilx = Iny; Ilx >= 0; Ilx-- ) {
        for( Iwx = 0; Iwx < ILen; Iwx++ ) {
            ITx = FSc * PFarray[Iwx] + FOff;
            if ( ITx < 0    ) ITx = 0;
            if ( ITx > Iny ) ITx = Iny;
            if ( Iwx == 0 ) IpTx = ITx;
            if ( (IpTx < Ilx && Ilx < ITx)  ||
                 (ITx  < Ilx && Ilx < IpTx) ||
                 (ITx == Ilx) )         Ich = 'x';
            else                        Ich = ' ';
            fputc( Ich, PFOutFp );
            IpTx = ITx;
        }
        fputc( '\n', PFOutFp );
    }
    return( 0 );
}
/* Convolve() - Convolve a filter with a one-dimensional signal */
int Convolve( PFilter, IFLen, PFInVec, PFResVec, ILen )
float           *PFilter;       /* pointer to filter coefficients */
int              IFLen;         /* number of coefficients in filter */
float           *PFInVec;       /* input signal vector */
float           *PFResVec;      /* output result vector */
int              ILen;          /* length of input & result vectors */
{
    int          IFx;           /* filter index */
    int          IResX;         /* result index */
    int          IResXLast;     /* index of last result item */
    int          IResXFirst;    /* index of first result item */
    double       DRv;           /* result value */
    /* --- check for things which do not make sense --- */
    if ( IFLen <= 0 || ILen <= IFLen )  return( -1 );
    if ( PFilter == (float *)0 ||
         PFInVec == (float *)0 || PFResVec == (float *)0 ) return( -1 );
    /* --- convolve the filter with the signal --- */
    IResXFirst = IFLen / 2;
    IResXLast  = ILen - (IFLen-1)/2;
    for( IResX = IResXFirst; IResX < IResXLast; IResX++ ) {
        DRv =  0.0;
        for( IFx = 0; IFx < IFLen; IFx++ )
            DRv += PFilter[IFx] * PFInVec[IResX-IResXFirst+IFx];
        PFResVec[IResX] = DRv;
    }
    /* --- handle left edge specially --- */
    DRv = PFResVec[IResXFirst];
    for( IResX = 0; IResX < IResXFirst; IResX++ ) PFResVec[IResX] = DRv;
    /* --- likewise right edge --- */
    DRv = PFResVec[IResXLast-1];
    for( IResX = IResXLast; IResX < ILen; IResX++ )  PFResVec[IResX] = DRv;
    /* --- we are done --- */
    return( 0 );
}

/* NNCycle() - perform one iteration with Neural Network */
int NNCycle( Bias, PFilter, IFLen, PFInVec, PFResVec, ILen )
float           Bias;           /* bias for PE */
float           *PFilter;       /* pointer to filter coefficients */
int              IFLen;         /* number of coefficients in filter */
float           *PFInVec;       /* input signal vector */
float           *PFResVec;      /* output result vector */
int              ILen;          /* length of input & result vectors */
{
    int          IFx;           /* filter index */
    int          IResX;         /* result index */
    int          IResXLast;     /* index of last result item */
    int          IResXFirst;    /* index of first result item */
    double       DRv;           /* result value */
    /* --- check for things which do not make sense --- */
    if ( IFLen <= 0 || ILen <= IFLen )  return( -1 );
    if ( PFilter == (float *)0 ||
         PFInVec == (float *)0 || PFResVec == (float *)0 ) return( -1 );
    /* --- convolve the filter with the signal --- */
    IResXFirst = IFLen / 2;
    IResXLast  = ILen - (IFLen-1)/2;
    for( IResX = IResXFirst; IResX < IResXLast; IResX++ ) {
        DRv =  -Bias;                                   /* NN special */
        for( IFx = 0; IFx < IFLen; IFx++ )
            DRv += PFilter[IFx] * PFInVec[IResX-IResXFirst+IFx];
        /* --- apply clamped linear transfer function to output --- */
        if (      DRv < 0.0 ) DRv = 0.0;                /* NN special */
        else if ( DRv > 1.0 ) DRv = 1.0;                /* NN special */
        PFResVec[IResX] = DRv;
    }
    /* --- handle left edge specially --- */
    DRv = PFResVec[IResXFirst];
    for( IResX = 0; IResX < IResXFirst; IResX++ ) PFResVec[IResX] = DRv;
    /* --- likewise right edge --- */
    DRv = PFResVec[IResXLast-1];
    for( IResX = IResXLast; IResX < ILen; IResX++ )  PFResVec[IResX] = DRv;
    /* --- we are done --- */
    return( 0 );
}

/* main() - main driver routine */
/* --- Input Signal --- */
float FSignal[] = {
  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,
  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,
  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,  0.15,  0.20,  0.25,
  0.30,  0.35,  0.40,  0.45,  0.50,  0.55,  0.60,  0.65,  0.70,  0.75,
  0.80,  0.80,  0.80,  0.80,  0.80,  0.80,  0.83,  0.80,  0.70,  0.90,
  0.80,  0.80,  0.60,  0.90,  0.40,  0.60,  0.30,  0.10,  0.20,  0.20,
  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,
  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,
  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,
  0.20,  0.20,  0.20,  0.10,  0.25,  0.30,  0.10,  0.20,  0.20,  0.20,
  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,  0.20,  0.20  };

/* --- Result Signal --- */
float FResult1[ ASize(FSignal) ] = {0};
float FResult2[ ASize(FSignal) ] = {0};
/* --- Convolver for Neural Network --- */
float FMHF[] = {
 -0.10, -0.60, -0.30,  0.50,  1.10,  0.50, -0.30, -0.60, -0.10 };
/* --- Standard (sobel) edge detector --- */
float FSobel[] = { -1.0,   0.0,   1.0 };

main()
{
    int          Iwx;
    float       *PFResA, *PFResB, *PFSwap;
    PrintGraph( &FSignal[0], ASize(FSignal), 40 );
    fputs( "\n--- Original Signal ---\n\n", PFOutFp );
    Convolve( &FSobel[0], ASize(FSobel),
              &FSignal[0], &FResult1[0], ASize(FSignal) );
    PrintGraph( &FResult1[0], ASize(FResult1), 40 );
    fputs( "\n--- Result of applying sobel edge detector to image---\n\n",
        PFOutFp );
    PrintGraph( &FSignal[0], ASize(FSignal), 40 );
    fputs( "\n--- Original Signal ---\n\n", PFOutFp );
    PFResA = &FSignal[0];
    PFResB = &FResult1[0];
    PFSwap = &FResult2[0];
    for( Iwx = 1; Iwx <= 8; Iwx++ ) {
        NNCycle( .02, &FMHF[0], ASize(FMHF), PFResA, PFResB, ASize(FSignal) );
        PrintGraph( PFResB, ASize(FResult1), 40 );
        fprintf( PFOutFp, "\n--- Cycle number %d ---\n\n", Iwx );
        PFResA = PFResB;                /* swap result pointers */
        PFResB = PFSwap;
        PFSwap = PFResA;                /* next ResB */
    }
    exit( 0 );
}
```

End Listing One

Listing Two

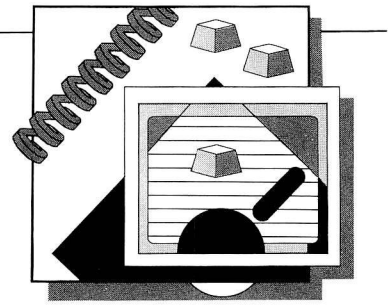
```
Neural Network Based "Edge Enhancement System"                          1
  Written by:  Casimir C. "Casey" Klimasauskas                          2
  January 6, 1990                                                       3
  Lotus 1-2-3 version 3.0 spreadsheet                                   4
                                                                        5
                                                                        6
                    0.020 Bias                                          7
                                                                        8
      Low    Low          Raw                                         9
     Pass   Pass    MHF   Input                                        10
   Output Filter Filter  Data                                         11
                                                                       12
Iteration                    0    1    2    3    4    5    6    7    8 13
Graph     A                  B    C         D         E              F 14
                                                                       15
                          0.20  0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 16
                          0.20  0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 17
                          0.20  0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 18
                          0.20  0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 19
```

(Listing continued on page 116)

Listing Two *(Listing continued, text begins on page 77.)*

A	B	C	D	E	F	G	H	I	J	K	L	M	Row / Column
0.00	0.00	-0.10		0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	20
0.00	0.00	-0.60		0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	21
0.00	0.00	-0.30		0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	22
0.00	-1.00	0.50		0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	23
0.00	0.00	1.10		0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	24
0.00	1.00	0.50		0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	25
0.00	0.00	-0.30		0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	26
0.00	0.00	-0.60		0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	27
0.00	0.00	-0.10		0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	28
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	29
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	30
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	31
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	32
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	33
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	34
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	35
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	36
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	37
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	38
0.00				0.20	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	39
0.00				0.20	0.03	0.02	0.00	0.00	0.00	0.00	0.00	0.00	40
0.00				0.20	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	41
-0.05				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	42
0.00				0.15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	43
0.10				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	44
0.10				0.25	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	45
0.10				0.30	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	46
0.10				0.35	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	47
0.10				0.40	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00	48
0.10				0.45	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00	49
0.10				0.50	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.00	50
0.10				0.55	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.00	51
0.10				0.60	0.04	0.00	0.00	0.00	0.00	0.00	0.00	0.00	52
0.10				0.65	0.05	0.00	0.00	0.00	0.00	0.00	0.00	0.00	53
0.10				0.70	0.09	0.00	0.00	0.00	0.00	0.00	0.00	0.00	54
0.10				0.75	0.15	0.12	0.18	0.28	0.46	0.76	1.00	1.00	55
0.05				0.80	0.17	0.20	0.31	0.49	0.79	1.00	1.00	1.00	56
0.00				0.80	0.15	0.12	0.16	0.26	0.43	0.71	1.00	1.00	57
0.00				0.80	0.10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	58
0.00				0.80	0.05	0.00	0.00	0.00	0.00	0.00	0.00	0.00	59
0.00				0.80	0.06	0.00	0.00	0.00	0.00	0.00	0.00	0.00	60
0.03				0.80	0.13	0.06	0.03	0.00	0.00	0.00	0.00	0.00	61
0.00				0.83	0.06	0.00	0.00	0.00	0.00	0.00	0.00	0.00	62
-0.13				0.80	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	63
0.10				0.70	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	64
0.10				0.90	0.21	0.08	0.06	0.01	0.00	0.00	0.00	0.00	65
-0.10				0.80	0.18	0.14	0.00	0.00	0.00	0.00	0.00	0.00	66
-0.20				0.80	0.22	0.05	0.00	0.00	0.00	0.00	0.00	0.00	67
0.10				0.60	0.13	0.05	0.00	0.00	0.00	0.00	0.00	0.00	68
-0.20				0.90	0.29	0.27	0.32	0.50	0.78	1.00	1.00	1.00	69
-0.30				0.40	0.26	0.28	0.38	0.58	0.93	1.00	1.00	1.00	70
-0.10				0.60	0.11	0.04	0.05	0.13	0.26	0.50	0.73	0.99	71
-0.50				0.30	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	72
-0.10				0.10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	73
0.10				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	74
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	75
0.00				0.20	0.05	0.04	0.03	0.02	0.01	0.00	0.00	0.00	76
0.00				0.20	0.01	0.02	0.02	0.02	0.01	0.00	0.00	0.00	77
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	78
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	79
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	80
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	81
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	82
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	83
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	84
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	85
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	86
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	87
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	88
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	89
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	90
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	91
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	92
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	93
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	94
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	95
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	96
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	97
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	98
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	99
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	100
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	101
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	102
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	103
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	104
0.00				0.20	0.01	0.02	0.02	0.01	0.00	0.00	0.00	0.00	105
0.00				0.20	0.06	0.04	0.02	0.00	0.00	0.00	0.00	0.00	106
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	107
-0.10				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	108
0.05				0.10	0.00	0.00	0.00	0.00	0.02	0.05	0.08	0.12	109
0.20				0.25	0.08	0.13	0.19	0.28	0.43	0.67	1.00	1.00	110
-0.15				0.30	0.12	0.14	0.20	0.29	0.45	0.71	1.00	1.00	111
-0.10				0.10	0.00	0.00	0.00	0.02	0.06	0.13	0.25	0.41	112
0.10				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	113
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	114
0.00				0.20	0.05	0.03	0.00	0.00	0.00	0.00	0.00	0.00	115
0.00				0.20	0.01	0.02	0.01	0.00	0.00	0.00	0.00	0.00	116
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	117
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	118
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	119
0.00				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	120
				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	121
				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	122
				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	123
				0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	124

End Listings



Wrapping Up Software Development '90

"Can you imagine — the Berlin Wall breached! Free elections in Poland and Czechoslovakia! The Soviet leader proclaiming freedom of religion while visiting the pope! The Romanian dictator getting frosted on Christmas Day! Bush found the whole sequence of events unbelievable, dizzying, even a little bit frightening — and yet, unquestionably, the fat lady had opened her mouth and was hitting high C."

— Jamie Malinowski

Michael Swaine

"Gee," the president marveled in his trademark nasal whine and 1950s sixth-grade vocabulary, "it seems like it was only yesterday that their Premier was making trouble all the time and saying he was gonna bury us and stuff, and now their whole darn evil empire looks like it's coming apart. Just look at what's happened with their Mac products and Turbo Modula-2 and Turbo Basic. And what was that business at the SD '90 conference about Wallsoft's C compiler? Why'd they let a competitor speak at their press conference? Well, they must have got some bad Perrier down in Scotts Valley.

"It's pretty neat for us guys, that's for sure," he told Steve "Wally" Ballmer, his squeaky-voiced confidante. "Now maybe the press will talk about that instead of always picking on us about our problems here at home or our problems with our so-called buddies at IBM." Blood brother oaths sworn in the childhood of the industry didn't seem to mean much in these grown-up days.

Chastened by this thought of the challenges that still lay before him, the president frowned at his long-time friend. "Gee. It's tough being big, Wally," he said.

Drugs, Bugs, and the DoD

Having joked about bugs as drugs in my February "Flames," I was interested to see that familiar *DDJ* author Do-While Jones was speaking about debuggers as a drug at Miller-Freeman's Software Development '90 in February.

"When you get sick," Jones said, "you should take medicine until you get well. Then you should stop." When your software is sick, he reasoned, you should use a debugger to find the problem. When you find the problem, you should put the debugger back on the shelf.

Debugger abuse, Jones says, comes from using a debugger when the program really isn't sick. Sick programs include those inherited in a messed-up state from someone else, and your own programs when they quit working suddenly, perhaps due to a change in the hardware or compiler or operating system. If something like this isn't wrong with the program, you shouldn't use a debugger, he says, because you will get hooked, starting a cycle of abuse: Dependence on a debugger makes for weak programmers who write sick software that can't be debugged without a debugger.

His point is that you usually don't need a debugger, that your knowledge, skills, and insight into the structure of the code constitute the best possible debugger. But you can't use your knowledge of the structure of code that has none. Writing well-structured programs is the prerequisite to rediscovering "the lost art of debugging."

As Jones describes it, the lost art is one of analyzing data flow. In a hierarchically structured program, modules at one level invoke modules at the next lower level, passing data to them, typically as parameters, getting data back, possibly as function values. Debugging a program consists of examining this data flow. The techniques he describes shouldn't be a revelation to any *DDJ* reader, but the kind of rigor he promotes in their use is perhaps uncommon.

You can test the data flow into and out of a module by writing a driver for the module. The driver sends selected data to the module and examines the results. This tests the module in isolation, but doesn't reflect its interaction with other modules.

For this, you need an integration test driver that calls the module one level higher than the modules you want to test. Data values are fed to it in order to check the interaction of the modules it invokes.

Finally, you can substitute a diagnostic module for any module in your program. A diagnostic module has the same name and parameters as the real module, but has a body that is simple and diagnostic. That is, it isn't so complicated that it could reasonably be a source of errors, and it does nothing but give useful information on the data-passing structure of which the module is a part.

In making his point, Jones used the word "hacker" to refer to one who pokes at the problem more or less at random until finding a fix that works. The word has a range of meanings, some good, some not, and this use is a legitimate use from the pejorative end of the word's meaning range. It was interesting, then, to find that he

espoused a distinctly hackerish philosophy in another talk at SD '90.

This was a discussion of the difference between engineers and computer scientists. Most of what he said was uncontroversial: Engineers are typically better educated in the physical sciences, computer scientists better grounded in a variety of programming languages. But the key differences, he said, are philosophical. Computer scientists and engineers have different ideas of economy; engineers are trained to look at the whole system, while computer scientists see the hardware as the platform, that is, as a given; computer scientists want to apply the best algorithm for the task (and are better equipped to do so), while engineers let the task dictate criteria for acceptability of an algorithm.

Jones is an engineer, and presented the engineering approach as the more hackerish, the more ad hoc of the two: Solve the problem no matter what. Here are some of the kinds of ad hoc, hackerish things Jones was saying:

- Most embedded systems don't need any kind of operating system.
- Data structures aren't important if you only have a few variables and RAM is limited.
- Tools are overrated.

Taken together, the talks make a point at a higher level. On the one hand, structure your code so that you can debug it by hand, and structure your approach to debugging. On the other, get the job done, meet the target launch date, satisfy the client. Together, the talks say simply, suit the tactic to the task. Every programmer should have a rich collection of ways of approaching a problem — a full set of intellectual tools. Structure is essential but so is flexibility.

Flexibility is a good trait for any software developer to cultivate. Do-While Jones is a Department of Defense engineer, like many who push bits for a living. One speaker at SD '90, William Roetzheim, speculated that half the software engineers in America were now or would be at some time in the future writing to DoD specs. Well. I don't relish the thought of anyone losing his or her job, but I am skeptical. This would not seem to be the best time to begin learning Ada.

The Big Challenge of Programming in the Large

"Stop talking of war cause we've heard it all before. Why don't you go out there and do something useful?"

— Sinead O'Connor

The image: The development team as bureaucracy, ideas trampled under political arguments, progress held up by mandatory progress reports, brilliant developers brought down to the level of their feeblest teammates.

The reality: The same thing, all too often. There appear to be excellent reasons to fear and to loathe the development team, and to long for the freedom to just write the damn thing. We all know the stories of the lone programmers who created masterpieces. It may be unrealistic to think that large software projects can be done any other way than through development teams, but it can be an inviting fantasy.

Knowledge, skills, and insight into the structure of the code constitute the best possible debugger

And yet, there are the stories of the team that worked. And most have even experienced that golden time, when the team fed off each others' talents and the result was a collaboration none of the participants could have done alone, and that all were proud of. OK, maybe it was only the Pringles can sculpture that we built on the Dean's front porch when we were freshmen, but when it comes to that sense of shared ownership of the work, is building Pringles can towers any different from writing for Presentation Manager?

What's The Secret?

The toughest problems usually turn out to be the ones involving other people. What we suspect seems to be true: Group dynamics may be the most important factor in software development group success. As Richard Cohen, speaking at SD '90, said "software development productivity is more strongly affected by people- and team-related issues than [by] any other variable under the management's control."

And teams are increasingly going to be where it's at. Programming in the large is — err — getting bigger. Ken Orr maintains that coding skill grows less and less important as the project gets bigger. "For people trained in good software engineering approaches, writing individual programs that aren't very

large or complex is not a critical skill. On the other hand, the planning, architecture, requirements, and design of large suites of data files and programs (programming in the large) are critical skills [and] will become more so. The need for software engineers will remain acute, but increasingly these people will be writing systems, not programs."

If programming in the large is the task of the future, team programming is the paradigm of the future. Cohen nailed down what may be the essence of how successful teamwork feels: "A real team," he said, "is one in which the group feels common ownership of the problem and its solution." Many speakers at SD '90 talked about problems involved in team programming, managing software engineering projects, programming in the large. I didn't catch all their talks, but after the conference I sifted through the proceedings and my notes, and found that many of them dealt with the search for ways to achieve that sense of common ownership of the work.

It's an important search. Changes are afoot, SD '90 speaker Tim Twinam says, and the customer may well start calling more of the shots. The free ride of the Trappist technician may not end this year, but there is an inherent instability that will some day shake out.

Vern Crandall and Larry Constantine have given some thought to the group structures conducive to good teamwork. Crandall, who worked at Novell, claims that fresh thinking is required in this area because most of the old answers, the software methodologies, including Orr's data-structured system development (DSSD), were developed for MIS development work, not for commercial software development. "We need a product-oriented approach," he said.

He lists some of the issues that are unique to or more important in commercial software development than in MIS:

- Ill-defined users (you can't walk down the hall and look over their shoulders to see what they're doing wrong);
- Programming to a moving target (the market changes during the development process);
- Multiple, complex models (the batch mode of early MIS is history; today a commercial product may have aspects of several models, including real-time operating system, embedded system, control software, distributed processing, communications, WAN, LAN, device driver, and database);
- Multiple industry standards and platforms;
- Many quality issues (reliability, installability, configurability, serviceability, usability, interoperability, performance, security, recoverability, and migration); and difficulties in customer education and support.

At least six groups of people must all work together well to make the project come off: Marketing, software development, software testing, software maintenance, documentation, and human factors. This is a complex, interrelated system of individuals with different skills and interests, all pushing to meet a common deadline that is invariably too tight. Crandall claims that flat and network management structures don't work in such an environment. He points out, furthermore, that software developers are creative people and that rules, regulations, and restrictions need to be kept to a minimum to avoid stifling their creativity. He hails the benefits of consensus management in maintaining the flexibility needed to let creative people solve problems together. But the flexibility and creativity can cause the developers to get off track, so a lot of supervision is necessary, as well as a lot of encouragement and direction.

He thinks that a hierarchical structure and an emphasis on accountability, challenge, and expectation is ideal.

How do you achieve this? Crandall has some suggestions: No individual team should be larger than six people. Products should be staggered, not sequentially released. Testing and maintenance are skills; hire testing and maintenance engineers, not testers, and pay them as much as the development engineers. Most of the existing models for a product life cycle are too long; you need to run in parallel as much as possible. In particular, software development, testing, documentation, and human factors (such as screen design) must run in parallel. This demands a lot of communication and a lot of freedom of movement, and that demands careful structuring of the teams.

People do care about quality and will work harder and better when they feel they are working on a product they can take pride in

Larry Constantine is a pioneer in the development of structured programming and structured design. He gave three talks at SD '90, and in one of them defined what he calls the "structured open team." It's apparently what Crandall is describing. "The structured open team uses formal structure to increase internal flexibility and adaptability while maintaining simple, external interfaces and behavior. Internally, it is structured to function as a tight-knit, closely integrated team of professional equals with clear differentiation of functions only as necessary for effective functioning." One of the key aspects of the structured open team is the default assignment of responsibility. Responsibilities can be shifted around as people's skills and knowledge and interests (and the changing demands of the job) require, but there is always a default assignment of responsibilities to ensure that nothing falls through the cracks.

One of the key steps in developing a software product is testing. Testing is naturally more complex in multiprogrammer projects, but Crandall turns up a surprising fact: Few schools offer any training in testing. Roetzheim, though, pointed out that testing is an important element of DoD specs. Maybe more programmers will be using DoD specs than I speculated earlier. Some companies today write to DoD specs in certain circumstances even when not doing Defense work.

But the most compelling point to come from these talks is that it is the human issues that are critical. Cohen listed some of the human traits that any software development team must deal with:

- People make mistakes
- People are often blind to their own errors
- People misunderstand each other
- People fixate
- People get overwhelmed by too many details
- People identify with their work
- People care about quality

The last point is a particularly good one for managers to keep in mind: People do care about quality and will work harder and better when they feel they are working on a product they can take pride in.

It was P.J. Plauger who sounded the contrarian note. There is always danger in accepting the common view uncritically, and there seems to be some sort of common view of good software management emerging (which is not to say that good software management is emerging). Plauger offered, for the stimulation of software management thinking, his contrarian list of software management heresies:

- Every software project must be just slightly out of control
- Your goal as a manager is to make software projects boring
- Your obligation to your programmers is to answer their telephone calls
- Your indispensable programmers are your greatest liability
- Teaching BAL programmers C++ is a waste of time
- Staying within budget is more important than making a profit
- Writing software must be fun, but not too much fun

Plauger justifies these heresies convincingly, but I think they are more useful if you are allowed to supply your own explanations.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 9.**

(continued from page 158)

HCR/C++, Version 2.2, is the first upgrade on this C++ compiler from **HCR Corporation**. HCR/C++ operates on most 386 and 486 Unix systems. This version includes class libraries that contain a comprehensive set of predefined objects that are compatible with the reusable object capabilities of C++. HCR/C++ comes with an enhanced release of dbXtra, the window-oriented debugger compatible with Berkeley's DBX.

HCR/C++ 2.2 includes the set of class libraries defined by AT&T and those in HCR/C++ 2.1, as well as the NIH (National Institute of Health) libraries, which provide classes for strings, linked lists, date and time conversions, indexed arrays, hash tables, regular expressions, and vector operations. The InterViews libraries that were developed at Stanford University and provide an interface to X Windows are also part of the package. The product should be available by now, and retails for $995. Version 2.0 customers can upgrade for $99. Reader service no. 23.

HCR Corporation
130 Bloor St. West, 10th Floor
Toronto, Ont. Canada M5S 1N5
416-922-1937

The bStrings Library for C, which adds dynamic string-handling capabilities to the C language, is available from **KBM Communications**. The bStrings Library provides dynamic strings without fragmenting memory. Over 130 string manipulation routines are provided, which duplicate most every string function available in Basic, as well as some not found in that language. The library supports functions that cut, copy, paste, clear, and overwrite whole strings or sections of strings, and will work with most screen management packages.

This library is available for Borland's Turbo C 2.0, Microsoft C 5.0/5.1 and QuickC 2.0. Both versions are provided with each order, and come with a 30 day money-back guarantee. The product sells for $89.95. Reader service no. 36.

KBM Communications, Inc.
2401 Lake Park Dr., Ste. 160
Atlanta, GA 30080
800-227-0303

A QuickBasic file indexing program, Index Manager, has been released by **CDP Consultants**. This product supposedly gives programmers the ability to create B+ tree files indexed within their QuickBasic programs. It allows random file access by full key, browsing through files by partial key, or sorting forward or backward. One external subroutine performs all of Index Manager's functions, and the programmer still retains full control over all data files, as only indexes are managed.

Indexes are created with a prefix B+ tree. The program is written in assembler language, and utilizes a large cache buffer for keeping important index records in memory. A demo version can be downloaded on CompuServe (GO MSSYS) on data library 1 or 2, called INDEXM.ARC, and GEnie (M 505) on data library 10, file 828. The program costs $59. Reader service no. 25.

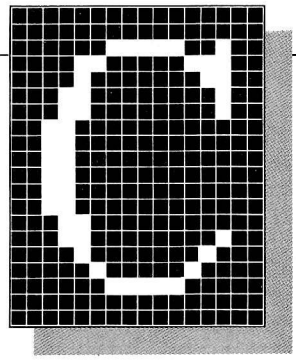
CDP Consultants
1700 Circo del Cielo Dr.
El Cajon, CA 92020
619- 440- 6482

New run-time tools are now available from **Gold Hill Computers** for its development environments GoldWorks II and GCLISP Developer 3.1. The company claims that these products will produce applications that can be invoked directly from DOS or Microsoft Windows/286, that they will load as much as five times faster than applications loaded under the development environments, and that they will require much less memory.

GCLISP Runtime supports the delivery of GCLISP Developer 3.1 applications. The Lisp run-time configuration requires 1 Mbyte of extended memory, and so can deliver applications with less than 2 Mbytes of memory. Goldworks II/PC Runtime configuration requires 2 Mbytes of extended memory, with 4 Mbytes memory targeted for end-user machines, and is integrated with external programs such as Lotus 1-2-3, dBase III, and C. Gold Hill's Starter-Pak is $1000 for GCLISP 3.1 and $1500 for Goldworks II/PC. Reader service no. 27.

Gold Hill Computers, Inc.
25 Landsdowne St.
Cambridge, MA 02139
617-621-3300

DDJ



CSORT: A Saga of a Sort

Al Stevens

Those of you with mainframe experience know that a common and indispensable utility program in that environment is the file sort program. Many batch applications involve the generation of reports taken from data files that you maintain in sequences other than the sequence needed for the report at hand. I remember the two- and three-way tape sorts on the IBM 1401 from as far back as 1961. It would have been unthinkable to try to design a system without one.

The Mainframe Tape Sort

Here's how such a sort works in the typical tape-drive configuration of yore. You need at least four tape drives to run a two-way sort. The unsorted file starts out on the first tape drive, and the other three have scratch reels of tape. A file of parameters (usually on a control card) tells the sort program the size of the file's records and the location, length, and sequence (ascending or descending) of each of the fields to be sorted.

Pass One, the Input Pass — The sort program reads as many records into memory as the CPU can hold and sorts them in an array. Then it writes the sorted block of records to the third tape drive. Next, it reads and sorts another block of records and writes that block to the fourth drive, then the next block to the third, and so on until all the input records have been read. Drives 3 and 4 now each contain multiple blocks of individually sorted records, and this is the end of the first pass.

During the first pass, the block size is restricted to the amount of memory that is available. Block sizes double during subsequent passes because each block is in sequence, and the program reads them a record at a time to perform the merge.

Pass Two, the Merge-down Pass — The computer operator replaces the input tape on drive 1 with a fourth scratch tape, and the sort program merges the first block from drive 3 with the first block from drive 4, writing the merged block onto drive 1. Then it would merge the next blocks from drives 3 and 4 onto drive 2. This flip-flop continues until all the blocks from drives 3 and 4 are merged into drives 1 and 2, which together now contain half as many blocks as 3 and 4. (The old-timers among my readers are beginning to get bored.)

Pass Three, the Output Pass — The merges continue back and forth with each pass reducing the number of ordered blocks by half and doubling the size of each block until at the final pass there is only one block on either drive 1 or 3, and this tape contains the sorted file, which can be read by the application.

Sordid Data on Micros

When I began working with microcomputers in the mid-seventies I looked for the standard sort utility program that I thought would come with operating systems, and I found none. The first significant application I wrote for an IMSAI 8080 needed file sorting, and a search turned up a CP/M program called "QSORT," but I was surprised to learn that it was difficult to find what had always been an indispensable utility program.

With the PC and MS-DOS came the SORT filter program, but it has three serious limitations: It can only sort as many records as will fit into 64K, it can sort only one field of the file's records, and, being a filter, it works only with text files.

The In-line Sort

So much for the file sort utility program. Now let's consider the in-line sort feature. Anyone who has programmed in many computer languages has accumulated some favorite features about each one. The ideal personal language would have all those features wrapped into one. Of course, when you try to build a language with everyone's favorite features, you get a committee-designed language — you get ADA, perhaps. The perfect personal language would be so extensible that each of us could design his own syntax and data types, plugging in the features we treasure and leaving out the ones we do not. The flaw in that approach is that no one could read anyone else's code, and so, instead of each of us building a personal ideal language, we ebb to the committee approach, standards emerge, and we strive to conform.

As a full-time C programmer and writer, I often think of language features in terms of the features I liked about languages past that C does not have. For example, the string functions of Basic are missing in C. Cobol's MOVE CORRESPONDING would be handy in C where the C structure assignment would assign only those members that have the same names. You can add both of these features to C by building appropriate C++ classes if you are using the C++ extensions to C, and so the dubious realm of the customized language looms again in the near future.

There is one feature I liked about Cobol that you can build in traditional C without extending the language other than with a few functions. That feature is the SORT verb and it is relevant to this month's rambling.

A Cobol program can pass records to the SORT verb one at time and then later retrieve those records in the sorted sequence. There are several advantages

to this technique. One is that the program does not need to prepare an unsorted file, exit to a sort utility program, and then execute a third program to process the sorted file. You can form each unsorted record from one or more sources just prior to sending it off to be sorted, and you can transform the sorted records into another format before doing anything with them. No intermediate file of uniquely formatted records is involved.

The C *qsort* function is similar to this approach except that it sorts an in-memory array, and so all your data records must fit into memory. A true in-line sort will accept as many records, one at a time, as you want to give it, and will return those records to you, one at a time, in sorted order.

So now we have defined two requirements missing from the C language and the microprocessor operating system environment. Why, after ten solid years of micro use, are these features not standard? Obviously, they are not widely needed for some reason, and that reason is, I guess, that the personal, inexpensive nature of the PC has redefined the way we design systems. Most PC programs today are interactive. If they involve multiple sequences of data files, they use indexed database managers or higher-level DBMS languages that deal with sorting in their own ways. OK, so most of the time we do not need a sort program or an in-line sort. But what about the few times when we do? Nothing else will do.

About five years ago, I tackled this problem by writing a C language in-line *sort* function for some programs for a video tape store. Later I used it as the sort engine for a stand-alone sort program that sorts disk files. I used pre-ANSI Aztec C for the IBM-PC and published the code in a book (*C Development Tools for the IBM PC*, Brady Books) in 1986. Since then I have reused that program many times in circumstances where I might have chosen a more accessible but less appropriate solution had I not already had this little sort program. Now that the ANSI standard definition of C is in place, it's time to update that program, so this month we promote it to the standard and publish it here.

CSORT

What follows is CSORT, a sorting facility that you can use from within your programs or from the command line. Although I developed it to use in CP/M systems and later in MS-DOS, it is not wired to either of those environments and should easily port to another platform where a standard C compiler is available.

The CSORT project has two parts, the in-line sort and the file sort program. To use the in-line sort in a C program, you describe the characteristics of the records to be sorted and send them, one at a time, off to the sort process. After you have sent the last of the records, you send a NULL and go about your business. When you are ready for the records in the sorted sequence, you call the sort program and it returns the records, one for each call. After it has returned the last of the sorted records, it returns a NULL. Your program is essentially the polar ends of a three-phase pipe.

You and CSORT pass records back and forth with pointers. CSORT makes a copy of the record you pass it, so you can safely reuse the space the record occupied. CSORT expects you to do the same because it might reuse the record space that is pointed to to be a previous pointer it returned.

CSORT requires that you sort fixed-length records, but the records themselves do not need to be in text format. The sort comparison uses the *strnicmp* function to compare fields, so they do not need to be null-terminated.

(continued from page 128)

The CSORT API

Following is the application program interface for CSORT. Listing One, page 144, is CSORT.H, which contains several control values for the sort. The NOFLDS global value specifies the maximum number of fields that can be involved in the sort of the record. The MOSTMEM and LEASTMEM values control the appropriation of a buffer from the heap for sorting. MOSTMEM is the amount you try for, and LEASTMEM is the least amount you will accept. I have them set to work within a small-model program on the PC.

Here are the prototypes and descriptions of the CSORT API functions.

int init_sort(struct s_prm *prms); To sort records, you must describe them. The *s_prm* structure contains the definition of the records to be sorted. It is defined in CSORT.H. You will declare the structure and initialize it as follows. Assign the record length to the *rc_len* member. The *s_fld* array of substructures will contain entries for each of the fields to be sorted. The *s_fld.f_pos* member contains the record position for the field. This value is relative to one. If there are fewer than NOFLDS fields, the terminal entry in this array will contain a zero value for this member. The *s_fld.f_len* member is the length of the field. The *s_fld.ad* member is the character "a" if the field is to be collated in ascending sequence and "d" if it is to be collated in descending sequence. The first field in the array is the major sort field; the last is the minor; all others are intermediate. So, if you need records in, for example, division number, department number, and employee number sequence, you will define the fields in that order in the array. With the array properly initialized, call the *init_sort* function. If the sort may proceed — if enough memory is available — the function returns zero. Otherwise it returns –1.

void sort(char *rcd); With the sort program properly initialized through *init_sort*, you may send records to be sorted by calling the *sort* function once for each record. Pass a pointer to the record, and the *sort* function will make a copy of the record. After you have passed the last record, pass a NULL pointer. This tells the sort function to finish the sort and prepare to return sorted records.

char *sort_op(void); To retrieve sorted records, call the *sort_op* function. Each call to it will return a pointer to the next record in the sorted sequence. After the last record comes back, the *sort_op* returns a NULL pointer.

void sort_stats(void); This function displays some of the values used in the sort function.

The FILESORT Program

Listing Two, page 144, is FILESORT.C, a program that uses the CSORT API to implement a stand-alone file sort program. You run it by entering the filename, record length, and field parameters on the command line. It uses the API in the manner just described.

CSORT Internals

Listing Three, page 144, is CSORT.C, which contains the in-line sort functions. The first function to discuss is *init_sort*, which you call to initiate sorting. It calls *appr_mem* to allocate a block of memory in which to sort, initializes the sorting parameters, and returns.

The *sort* function accepts records to sort from the calling application. At first it simply copies them into the buffer, one after the other. When the buffer is full, the *sort* function calls the standard *qsort* function to sort the records in the buffer. This becomes a block of sorted records, also called here a "sequence." The *dumpbuff* function writes them to the sort work file. CSORT uses only

(continued from page 130)
one sort work file, unlike the tape sorts we discussed earlier. Because this program uses disk files, and because direct record addressing is possible, we do not need to maintain multiple, serial devices for the blocks of sorted records after the fashion of a streamer tape device.

When the calling application sends a NULL pointer to the *sort* function, it sorts the final sequence, writes it to the work file, and calls *prep_merge* to prepare for when the caller wants sorted records. The merge divides the sort work buffer into many miniature buffers, one for each sequence that was generated by the *sort* function. These buffers contain two or, usually, more records. If and while the number of sequences is high enough to prevent the buffer from being segmented this way, *prep_merge* uses the *merge* function to merge groups of two sequences into one, halving the number of sequences and doubling their sizes.

When the number of sequences is low enough that each one can contribute at least two records in the sort buffer, the *merge* and *prep_merge* functions are done.

The calling application calls *sort_op* to get sorted records. The *sort_op* function looks at the first record in each buffer segment to find the lowest record. It bumps that segment's record pointer and returns a pointer to the record it found. When it exhausts a buffer segment, it reads more records from the associated sequence on the sort work file. When all the segments are empty, the application has read the last sorted record, and the sort program signals that it is done by returning a NULL pointer.

TopSpeed C

Last year I reported that I had seen a demo of the announced TopSpeed C compiler from Jensen & Partners International. The product is shipping now, and is a full-featured ANSI-conforming compiler with integrated editor, compiler, linker, and debugger. I used it to upgrade CSORT to the ANSI specification and can offer these first impressions. Be advised that I have not yet used TSC to build a significant system from scratch, so these judgments are preliminary. I think you will like this product. After reading and following the installation procedures, I used the books only once during the project, and that was to learn the format for the MAKE project file. Everything else is neatly supported with comprehensive help windows and intuitive menus.

TSC has a number of features that I haven't used yet but surely will. They have built a DOS equivalent of the OS/2 dynamic link library (DLL). You can build programs that link to their libraries at run time. There is a post-mortem debugger, an assembler and disassembler, a program profiler, and support for DOS Windows and OS/2 Presentation Manager program development.

TSC has a few things I'd change. Their version of the *interrupt* function type has a severe disability. You cannot call an interrupt function directly or through a function pointer. This prevents you from chaining intercepted interrupts and renders TSC ineffectual as a TSR development compiler — unless you want to compensate by using assembly language.

Although TSC has the _AX, _BX, etc. pseudoregisters of Turbo C, they do not support _FLAGS, which makes their use in interrupt service routine programming even more difficult.

The tables of contents and indexes in most of the documents do not agree with the actual page numbers. This is an unacceptable lapse in quality control for any documentation effort. I forgive that lapse only because the on-

line documentation is so good.

TSC tends to leave my cursor other than the way it found it, and it leaves all manner of what appears to be temporary work files scattered about my source directory. These quirks are annoying at best.

One last gripe, then I'll relax. I rejoiced when I saw the WATCH utility. It is a really neat TSR, tossed in for good measure, that you can use to monitor the calls your program makes to DOS, an essential tool for systems programmers. I do a lot of programming in the Novell NetWare local area network environment, and NetWare API calls are supersets of the DOS INT 0 x 21. To my dismay, I learned that WATCH only watches calls that you select from its list of known DOS calls, and there seems to be no way to tell it to watch the Novell superset list. Rats. Another idea well conceived, poorly built.

If it seems that I am overly critical it is because these are the things I found that I think you should know about. There may be others, but, as I said, my use of TSC has been limited so far. All that aside, I like the product and do not hesitate to recommend it. It is in great shape for the first version of any software product and will surely improve with newer versions. The PC programmer has a dilemma. There are several excellent compilers to choose from. By now, there will have been benchmarks and reviews and, I am sure, you still will not know what to do.

(The title of this month's column is a paraphrase of that of the Bill Mauldin book, *A Sort of a Saga*. His book has nothing to do with C or programming, but Mauldin's cartoons and writings and that book in particular are timeless and were major influences in my young life.)

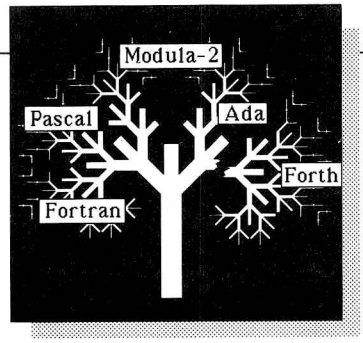
Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 144.)

Vote for your favorite feature/article.
Circle Reader Service **No. 10.**



Yes, Virginia, There Is a Xerox!

by Jeff Duntemann KI6RA

I damned near choked on my Cheerios the morning the *Mercury News* announced that Xerox was suing Apple over the Macintosh UI copyrights. Land o' Goshen! There is a Xerox after all! I had just about given up hope.

I have reason to feel specially about this whole situation. I spent ten years at Xerox (1974 – 1984) when much of their seminal research into user interface concepts took place. I was not directly involved with that research, but I spoke regularly with PARC people who were. I played with Smalltalk on the Alto prototype workstation, and I was one of the very first in Xerox MIS/DP trained on the Star workstation in early 1981. Unfortunately, I was one of the very few who ever saw the Star, much less worked on it. Had the Star caught on, Apple's suit against Microsoft would have been laughed out of court.

The Apple vs. Xerox Equation

For those of you who have been living in a Dewar's flask since 1985, let me recap: Last year Apple sued Microsoft and Hewlett Packard, claiming that Microsoft Windows 2.0 and HP New Wave violated Apple's visual copyrights on the Macintosh user interface. Earlier this year, parts of the suit concerning contractual technicalities were dismissed by the judge, leaving only the very large question of whether Windows 2.0 and New Wave indeed infringed on the Macintosh visual copyright.

Now Xerox, in its suit, claims that Apple appropriated the visual copyrights expressed in the Star workstation before the Macintosh was born, and that Xerox, as owner of said copyrights, is due royalties from the use of its interface.

There is a wonderful word to apply here, as ancient and ineluctable as fate itself: Checkmate. No matter which way Apple moves, it loses. Why? Because if Windows 2.0 infringes on the Macintosh, then the Macintosh infringes on the Star. I had to laugh at Apple's immediate protest that it was protecting its expression of the ideas pioneered by Xerox, and not the ideas themselves. Any three-legged salamander blind in one eye could tell you that Windows 2.0 and the Macintosh UI are not identical expressions, but instead are separate expressions of a common philosophy of user interaction. And if the similarities of expression between Windows 2.0 and the Mac are close enough to infringe on the Mac, then the Mac is similar enough to the Star to infringe upon the Star. Poof! Apple loses its claims of ownership.

It gets better. If Apple changes its mind and withdraws its Windows suit, Microsoft and HP have excellent grounds to complain that Apple brought the suit merely to stifle competition, leaving it open to prosecution under antitrust laws.

Couldn't happen to a nicer bunch of guys, eh?

Many people are saying that Xerox can't win its suit against Apple (or anyone else), because it waited far too long to bring the suit to begin with. This is true, but it may not matter. The scrutiny that Xerox will bring to bear on the whole question of who authored what and when will finally lay to rest the pernicious lie that Apple has some kind of ownership of anything with pull-down menus and overlapping windows.

Most people haven't seen the Star, so public opinion hasn't really gone up against Apple. I was there, however, when it happened, and I don't lie about these things: The Xerox Star is closer to the Mac than Windows is. That's the golden equation whose solution will, with luck, put windowing environments in the public domain and end this damned foolishness now and forever.

Thinking About Object Design

Just because you can swing a hammer doesn't mean you can design a house. That's the message that seems to be emerging from our first year as Object Pascal programmers. People are catching on to the syntactic nuances of object-oriented programming. Creating objects and methods is no longer quite the mystery it was in 1989. What does seem to be a mystery is the big picture of it all, containing sizeable questions like, "What should be an object? How do you divide your application into objects, and how do you apportion functionality to the objects in an object hierarchy?" This is the old architecture vs. carpentry duality made new again by the appearance of mass-market OOP languages. So put down that hammer for a bit, and let's talk about larger issues.

What Should be an Object?

Just after Turbo Pascal 5.5 and Quick Pascal appeared last May (almost simultaneously) people predictably began jumping on the OOP concept and taking it to absurd lengths, just for the exhilarating fun of it all. I recall hearing about guys who were creating objects out of ASCII characters, or even integers, because it seemed like the politically correct thing to do at the time.

As the vanguard soon discovered, making Pascal integers and characters into objects bought them almost nothing, and cost like hell. The first lesson we learned about OOP melted into conventional wisdom in record time: Atoms are atoms for a reason. Objects are a means of organizing complexity. Don't go looking for complexity where there isn't any. Molecules have to be made out of something.

My own rule of thumb comes down to this: Don't make objects out of anything that the compiler special-cases. This includes all fundamental types such as Boolean, Char, all numerics, and strings. In a language like Modula-2

where strings have no special status to the compiler, a string object might make sense. Certainly a long string type (in which you can keep more than 255 characters) is a prime candidate for objecthood. Strings, however, have special privileges with respect to text files, *Readln* and *Writeln*, and a whole slew of built-in string-handling routines. Furthermore, strings may be returned as function results, a capability I would be extremely reluctant to give up. Encasing a Pascal string within an object wrapper actually adds complexity to an application, and that's not how the game is supposed to be played.

An Exercise in Modeling Reality

When I wrote the tutorials in the Turbo Pascal 5.5 *OOP Guide*, this sort of question hadn't occurred to me, as I was nearly as new to Object Pascal as the rest of the PC world and was running but a step and a half ahead of the Comprehension Wolves through the whole project. What I knew of objects came to me from Smalltalk, in which everything is an object, not excluding atomic particles such as characters and integers. But my Smalltalk experience kept me in good stead when I had to confront questions from readers like, "How should I divide an application up into objects?" and, "How do I define the boundaries between objects?" There's no easy answer this time, but I've worked with a principle that I consider very effective: Whenever possible, reflect in your objects the divisions and relationships seen in reality. In other words, make your OOP programs simulations even when they're not really simulations.

There I go again, getting Eastern on you. Time to Get Real.

Artifacts, Not Actions

Objects were originally conceived by Simula's architects as models of elements of reality. Reality is not soup. Reality is lumpy, and the lumps, seen apart, are identifiable things with identifiable functions. A lamp is a thing that emits light when you hit its buttons correctly. A carpet keeps the oak floors from getting scraped. Your Peter Norton mug is a mechanism that cooperates with gravity to keep your coffee out of your lap. Every lump in reality has a name and a job, though the necessity of some, like city planners and the rock band Metallica, is seriously open to question.

All of this is second nature to anyone who has survived his or her Terrible Twos, but I remind you of the nature of reality because I want you to apply it to the way you arrange the concept of a program in your mind. Too many programmers make their programs like you'd make soup, by tossing in a pinch of this and half a cup of that and stirring it around until it becomes relatively uniform in color and texture and ceases to be lethal.

Because programming has always been an action-oriented process, programmers often begin by asking what steps constitute the larger action of the program. If you have gotten to this point, the cooking has already begun, and you're halfway to soup already. Never forget this: A program is not an action. A program is an artifact. Once you throw the artifact into your conceptual duck-press and squeeze it down to a statement of action, you've lost a great deal of valuable information about the idea of the program. To recap in a slightly different way: A statement of what a program does is far less useful than a statement of what the program is. The best example has been staring you in the face for years. Tell me first what a spreadsheet program does, and then when you've given up on that,

tell me what a spreadsheet is. A spreadsheet does a whole host of things, but what a spreadsheet is is a model of the classic accountant's paper ledger sheet. By now, the lights should be coming on.

Collecting Stamps

Seeing artifacts where you used to see actions is perhaps the most important skill to be learned in picking up OOP. Let's practice a little by identifying a program artifact and giving it reality as an object.

The artifact in question is the humble moment, the point in time at which something happens. In certain kinds of programming, particularly business or financial programming, the time and date when a transaction happens can be almost as important as the transaction itself.

You've seen evidence of a moment in every DOS directory listing: The date and time when the file was last modified. Every directory entry on a DOS disk has a time stamp and a date stamp, which are sometimes lumped together and simply called the time stamp.

That's your artifact. To avoid confusion between its time and date components, (because it incorporates both) I call it a "when stamp." Any time you need to retain an expression of the moment that something happened in your program, you can create a when stamp object for it.

Objects embody both data (the object's "state") and the actions necessary to manipulate that data. The data in a when stamp is straightforward: Some single, unambiguous expression of both clock time and calendar date.

DOS has such an expression in its directory time and date stamps. It makes sense to use what DOS uses, (especially when DOS can be made to do some of our work for us) so let's agree to keep our when stamp as DOS-compatible as possible. The DOS time stamp and date stamp are both 16-bit words, so we can combine them into a single 32-bit type in Object Pascal. The obvious candidate is *LongInt*, the 32-bit long integer type.

Having the when stamp stored in a single numeric type carries the side benefit that two when stamps can be compared for time priority just by comparing with the numeric greater than and less than operators.

Time values are never negative, so you might begin to worry about the sign bit in a long integer, which is a signed type. As it happens, you will have to worry, but not for a very long time. (More on this later — nonetheless, I still wish that Pascal had a *LongWord* type.)

Once we get the expression of time and date, we'll need methods to manipulate it. But before you start thinking about methods, get your data and its representation in order.

Encoding Time and Date, DOS-style

Understanding how our When object works depends heavily on knowing how DOS encodes its time and date stamps in their 16-bit words.

If we used neat decimal star dates like Captain Kirk, we'd be better off, but alas, a date is a set of three separate numeric quantities: Year, an ascending value that never repeats, month, a value that repeats regularly through a cycle of 12, and day, a value that repeats through a cycle of 28, 29, 30, or 31. This makes date arithmetic ugly unless the date is encoded as a single numeric value. The means is this: Express year, month, and date as binary quantities, then line them up in a single 16-bit word such that the year portion occupies the highest-order bits, the month portion occupies the next-highest order bits, and the day portion occupies the lowest-order bits. Done this way, two date stamps with different years will always compare correctly regardless of month or date; two stamps with identical years will always compare correctly on the basis of month, regardless of date, and so on. Which bits relate to year, month, and date is shown in Figure 1.

Note that the year is encoded in a peculiar (and I think unfortunate) way: As an offset from the year 1980. 1980 is thus year 0, 1981 year 1, and so on. This allows the year to be encoded in only a few bits, leaving plenty of room for the month and day in a 16-bit word. The downside is that you can't encode your birthday as a time stamp, since you were probably born considerably prior to 1980. (I was, and have the hairline to prove it.) Seriously, this limits the use of when stamps to things that occur in true calendar time while the computer is in use, and not to encode events that happened long ago. I was a little concerned about the use of a signed type to hold the when stamp. At some point, the bit encoding of the time and date will set the high bit in the long integer, turning the value negative in the eyes of the run-time library. This blows any possibility of valid comparisons out the window, because a later value that is negative will be seen as less (and hence earlier) than any positive stamp. However, when I did the math I found that the stamp does not turn negative until December 31, 2043, by which time I will either be dead or perfecting zero-G lovemaking techniques out past the orbit of Mars.

17 Pounds of Kitty Litter in a 16-pound Bag

The DOS time stamp presents a problem. No matter how you encode the hours, minutes, and seconds (and forget hundredths here) you end up with a minimum of 17 bits: Six for minutes (0 – 59); six for seconds (0 – 59), and five for hours (0 – 23).

You can usually cram 17 pounds of kitty litter into a 16-pound bag by shaking things around a little. Not so in the bit game — we pack 'em tight. Something has to give a little, so what we do is ignore every other second. This cuts the number of bits required to encode seconds to five, and 16 bits will suddenly hold the stamp. See Figure 2.

Note the way the stamp is encoded

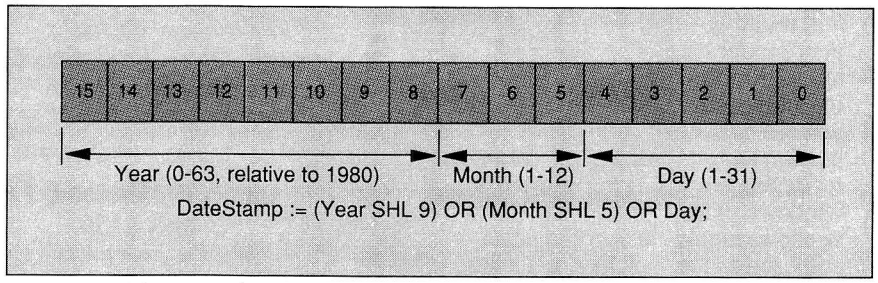


Figure 1: *The DOS date stamp format*

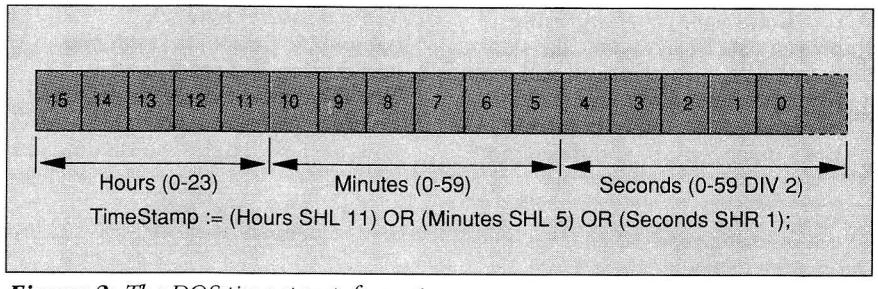


Figure 2: *The DOS time stamp format*

(continued from page 138)
in the figure: Whereas the hours and minutes values are shifted left to move them toward the high end of the word, the seconds value is shifted right by one bit, which bumps that bit off into Peoria and out of our hair. Truncating the seconds by half means that two events occurring less than two seconds apart will probably resolve to identical time stamps, depending on their synchronization with the system clock. You have to keep this in mind and avoid designing time stamps encoded this way into applications where stampable events happen in quick succession. Note also that your seconds value will always be an even number, because the 1 bit that can make it odd is not stored in the stamp.

To Represent or Calculate?

At the heart of it, then, a *When* object consists of a long integer time/date stamp encoded as explained earlier. If you only needed to compare two stamps for time order, this would be enough. However, you may need to access the seconds value separately, or the hours or months, and you may want to display time or date or both in some generally understood ASCII form. There are two ways to do this:

- Add separate fields to the object to contain distinct hours, minutes, and seconds; and years, months, and days. Then recalculate and update those fields any time the stamp itself changes.
- Leave the time/date stamp as the sole data field in the object, and calculate any component value whenever a user of the object requests it.

The first option buys speed at the expense of space, and the second buys space at the expense of speed. Which do you use? Well, do you need more speed or space?

I'm not just being glib here. There's no single answer. You as the object's designer have to keep track of your own particular needs. I tend to write small programs, and I like fast ones, so my own first impulse is to throw memory at problems to make them faster. This is the design choice I made in implementing the *When* object, as shown in Listing One, page 150.

The *When* object has as its central data item the *WhenStamp* long integer field. It also has separate fields for year, month, day, day of week, hours, minutes, and seconds. I added an ASCII representation of time identical to that used in the DOS DIR command, as well as two ASCII date representations: One in the form *yy/mm/dd*, and the

other in the form "Thursday, June 29, 1989." This adds 94 bytes to the size of the object, but I did it with eyes open and decided that the benefits were worth the cost. There was one exception. I chose not to provide a separate 16-bit time and date stamp, as I had originally considered doing, because returning one half or the other of a long integer can be done without any significant calculation overhead, just by typecasting. Notice the definition of *WhenUnion*, private to the unit:

```
WhenUnion =
   RECORD
      TimePart : Word;
      DatePart : Word;
   END;
```

WhenUnion is the same size as *LongInt*, so you can use a value typecast to access either the time or date portion of the combined time/date stamp:

```
TimeStamp :=
  WhenUnion(WhenStamp).TimePart;
```

It's that easy, and saves you 4 bytes of memory at the cost of no calculation at all. I like that sort of deal — kind of like insider trading in Silicon Valley.

Choosing Your Methods

Once you've decided what your object is, you can begin working out what it does; that is, design its suite of methods. I wanted my when stamp object to be easily updated, so I provided numerous methods for changing the value of the *WhenStamp* field. All the methods beginning with *Put* change the value of *WhenStamp*, and all the methods beginning with *Get* return some value from the object. This is a good naming convention when designing objects, and I recommend it. The *PutNow* method is simple but extremely useful: It reads the current value of the PC's clock/calendar and applies the current time and date to the when stamp. The other *Put* methods alter the value of the when stamp by providing a new value for either the whole stamp (*PutWhenStamp*) or some component value of the stamp, such as year, month, or hours. How many such methods you build into an object depends on how you intend to use the object. My recommendation is to build more into the object than you may need, and reexamine the design some time down the road. You can always strip out what you haven't used . . . but you never know when some flash of insight will allow you to find a use for a method that you hadn't imagined when you originally wrote it!

The Encapsulation Question

The "pure" object-oriented languages such as Smalltalk and Actor both allow and enforce total encapsulation of an object's data. You cannot directly reference an object's data from outside the object or its descendants. Object Pascal allows total encapsulation, in that you can voluntarily provide a separate method to return the value of each field in the object. Nothing enforces this, however — you can reference any field in any object from anywhere in your program. Purists will say, "So what?" Give 'em the methods anyway, and say hands off the data. Such hardnosedness gives you the considerable benefit of being able to change the actual representation of the data within an object without breaking the code that uses the object. For example, if I forbade direct references to the data inside *When*, I could come up with my own time stamp format that was truly universal and did not ignore the existence of years prior to 1980 as well as every other tick of the clock.

If you needed to provide stamps for old financial records, birthdays, and such (as in a life insurance application) you'd be far better off going this route. My own need for time stamps, however, was limited to stamping transactions occurring at the current moment, so I made the decision not to add the additional level of complication to When. I also wanted to retain full compatibility with the DOS directory time and date stamps, and the easiest way to do that is simply to represent the time and date the same way DOS does. Given these assumptions, I created When to allow direct read access to all the data fields.

Note that I said read. Writing directly to the fields is not a good idea, because all the fields but *WhenStamp* are actually component values of *WhenStamp*, and if you were to change the *Month* field without changing *WhenStamp*, the two would be "out of sync." The Put methods were designed so that changing any part of the time stamp recalculates all component values relating to that part of the time stamp. For example, changing the date half of the when stamp recalculates the *Year*, *Month*, *Day*, *DayOfWeek*, *DateString*, and *LongDateString* fields, without affecting *Hours*, *Minutes*, *Seconds*, or *TimeString*.

You need to be aware of such dependencies when deciding how to allow updating of an object's data. Calculating component values from a master field such as *WhenStamp* whenever they are needed avoids problems like this, if you can tolerate the loss of performance.

Private Parts

For reasons unknown, many newcomers to OOP get the notion in their noggins that all of an object's processing must be confined to its methods, and that while methods can call one another, methods somehow cannot call other procedures and functions unrelated to the object. Not true! A method can call any routine within its scope, just as any nonmethod procedure or function can. Furthermore, there is no way to declare a "private" method in Object Pascal. A private method would be one that could be called by other methods within the object, but that could not be accessed from outside the object. If there is any private processing to be done by an object, you can do what I've done with When and place the object definition in a unit — with private procedures and functions fully declared and defined within the implementation section. Such procedures and functions may be called by the object's methods but are not available to anyone outside the unit itself. Similarly, the *MonthTags* and *DayTags* constants exist for the convenience of the object and are not needed by users of

the object, so I've declared them privately, within the implementation section. Ditto with *WhenUnion*. As long as you provide functions to return separate 16-bit values for the time stamp and date stamp portions of the when stamp (as I did with *GetTimeStamp* and *GetDateStamp*), there's no need to give the user the *WhenUnion* definition. Half the battle of programming is masking complexity, so keep an object's private parts under a bushel basket where they won't be stepped on or misused by the ignorant and the unwary.

One of the private routines is a day-of-the-week calculator using an algorithm called "Zeller's Congruence." I've had this routine in my files for so long I no longer remember who coded it up and gave it to me, and I categorically do not understand how it works . . . but it does work.

Homework Assignments

That's how I went about designing a very simple object. To make sure it sinks in, do the following things for practice:

- Recode the *When* object such that the only data field is the long integer *WhenStamp*, with all other component values and alternate representations calculated as they are needed, by new *Get* methods. This shouldn't involve much new code at all, but will involve moving existing code around a lot. While you're at it, you might add a simple function to return the full long integer value of *WhenStamp*, giving you total encapsulation. You might like this more compact version of When more than mine. So be it. I'm not you. (And I suspect we both should be glad. . . .)
- Write new methods to add or subtract specific values from the when stamp. You might want to create a when stamp containing a value exactly 30 days from right now, and then monitor that stamp over the coming month to ensure that something necessary happens at that moment. This requires a way of adding 30 days to the long integer *WhenStamp* value. It's fairly easy . . . do it!

Object Design Recap

To summarize:

An object is an artifact, not an action. Look at the way the universe is broken down into components, and learn to think of your programs as built of component parts in much that same way.

Once you've identified such a software artifact, decide how to represent its data first. Only once you have the data design down should you begin to ponder what methods it needs to serve that data. Remember, Data is Boss in object land.

It's possible and often desirable to disallow any direct access to an object's data. This frees you to change the way the data is stored within the object without breaking code that uses the object. Remember, however, that this can add significant performance handicaps to code that uses your objects. Keep the tradeoffs in mind, but (as the cricket kept saying) always let your conscience be your guide.

Also remember that this is just a first lesson. We haven't even begun to address the issues presented by inheritance or, lord knows, polymorphism. All in good time.

Have You Seen this Book?

Maybe you all can help me out a little. My newest book, *Assembly Language From Square One*, (Scott, Foresman & Company) has been off the presses for some time, and I have yet to see it in any store. If you have seen it anywhere, drop me a postcard at *DDJ* and tell me what store is carrying it. The computer book distribution system seems to be breaking down in recent months, much as it did in 1984. Huge numbers of titles, most fit only to hang in a Tennessee outhouse, have been pouring into the retail channel lately, and it's gotten me (and some other well-known authors) more than a little worried. We may not be able to change the system, but we'd at very least like to know what books are going where.

Thanks.

Write to Jeff Duntemann on MCI Mail as JDuntemann, or on CompuServe to ID 76117,1426.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 150.)

Vote for your favorite feature/article.
Circle Reader Service **No. 11.**

Listing One *(Text begins on page 127.)*

```
/* ---------------------- csort.h --------------------------- */

#define NOFLDS 5        /* maximum number of fields to sort   */
#define MOSTMEM 50000U  /* most memory for sort buffer        */
#define LEASTMEM 10240  /* least memory for sort buffer       */

struct s_prm {              /* sort parameters                */
    int rc_len;             /* record length                  */
    struct  {
        int f_pos;          /* 1st position of field (rel 1)  */
        int f_len;          /* length of field                */
        char ad;            /* a = ascending; d = descending  */
    } s_fld [NOFLDS];       /* one per field                  */
};

struct bp   {           /* one for each sequence in merge buffer */
    char *rc;           /* -> record in merge buffer          */
    int rbuf;           /* records left in buffer this sequence  */
    int rdsk;           /* records left on disk this sequence    */
};

int init_sort(struct s_prm *prms);  /* Initialize the sort    */
void sort(char *rcd);               /* Pass records to Sort   */
char *sort_op(void);                /* Retrieve sorted records*/
void sort_stats(void);              /* Display sort statistics*/
```

End Listing One

Listing Two

```
/* --------------------- filesort.c ------------------------ */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "csort.h"

/* sort a file:
 *     command line: filesort filename length {f1, l1, az1, ...}
 *     filename is the name of the input file
 *         length is the length of a record
 *         for each field:
 *             f1 is the field position relative to 1
 *             l1 is the field lengths
 *             az1 = A = ascending, D = descending
 */

static struct s_prm sp;
static void usage(void);

void main(int argc, char *argv[])
{
    int i, fct = 0;
    FILE *fpin, *fpout;
    char *buf;
    char filename[64];

    /* ------- get the file name from the command line ------ */
    if (argc-- > 1)
        strcpy(filename, argv++[1]);
    else
        usage();
    /* ----- get the record length from the command line ---- */
    if (argc-- > 1)
        sp.rc_len = atoi(argv++[1]);
    else
        usage();
    /* ----- get field definitions from the command line ---- */
    do  {
        if (argc < 4)
            usage();
        sp.s_fld[fct].f_pos = atoi(argv++[1]);
        sp.s_fld[fct].f_len = atoi(argv++[1]);
        sp.s_fld[fct].ad = *argv++[1];
        argc -= 3;
        fct++;
    } while (argc > 1);

    printf("\nFile: %s, length", filename, sp.rc_len);
    for (i = 0; i < fct; i++)
        printf("\nField %d: position %d, length %d, %s",
            i+1,
            sp.s_fld[i].f_pos,
            sp.s_fld[i].f_len,
            sp.s_fld[i].ad == 'd' ?
                        "descending" : "ascending");

    if ((fpin = fopen(filename, "rb")) == NULL) {
        printf("\nInput file not found");
        exit(1);
    }
    if ((buf = malloc(sp.rc_len)) == NULL ||
              init_sort(&sp) == -1)   {
        printf("\nInsufficient memory to sort");
        exit(1);
    }
    /* ------ sort the input records ------- */
    while (fread(buf, sp.rc_len, 1, fpin) == 1)
        sort(buf);
    sort(NULL);
    fclose(fpin);
    /* ----- retrieve the sorted output records ------ */
    fpout = fopen("SORTED.DAT", "wb");
    while ((buf = sort_op()) != NULL)
        fwrite(buf, sp.rc_len, 1, fpout);
    fclose(fpout);
    sort_stats();
    free(buf);
}

static void usage(void)
{
    printf("\nusage: filesort fname len {pos length ad...}");
    exit(1);
}
```

End Listing Two

Listing Three

```
/* ---------------------- csort.c ------------------------- */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "csort.h"

static struct s_prm *sp;  /* structure of sort parameters    */
static unsigned totrcd;   /* total records sorted            */
static int no_seq;        /* counts sequences                */
static int no_seq1;
static unsigned bspace;   /* available buffer space          */
static int nrcds;         /* # of records in sort buffer     */
static int nrcds1;
static char *bf, *bf1;    /* points to sort buffer           */
static int inbf;          /* variable records in sort buffer */
static char **sptr;       /* -> array of buffer pointers     */
static char *init_sptr;   /* pointer to appropriated buffer  */
static int rcds_seq;      /* rcds / sequence in merge buffer */
static FILE *fp1, *fp2;   /* sort work file fds              */
static char fdname[15];   /* sort work name                  */
static char f2name[15];   /* sort work name                  */

static int comp(char **a, char **b);
static char *appr_mem(unsigned *h);
static FILE *wopen(char *name, int n);
static void dumpbuff(void);
static void merge(void);
static void prep_merge(void);

/* -------- initialize sort global variables----------  */
int init_sort(struct s_prm *prms)
{
    sp = prms;
    if ((bf = appr_mem(&bspace)) != NULL)   {
        nrcds1 = nrcds = bspace / (sp->rc_len + sizeof(char *));
        init_sptr = bf;
        sptr = (char **) bf;
        bf += nrcds * sizeof(char *);
        fp1 = fp2 = NULL;
        totrcd = no_seq = inbf = 0;
        return 0;
    }
    else
        return -1;
}

/* --------- Function to accept records to sort ------------ */
void sort(char *s_rcd)
{
    if (inbf == nrcds)  {   /* if the sort buffer is full */
        qsort(init_sptr, inbf,
            sizeof (char *), comp);
        if (s_rcd)  {   /* if there are more records to sort  */
            dumpbuff(); /* dump the buffer to a sort work file*/
            no_seq++;   /* count the sorted sequences         */
        }
    }
    if (s_rcd != NULL)  {
        /* --- this is a record to sort --- */
        totrcd++;
        /* --- put the rcd addr in the pointer array --- */
        *sptr = bf + inbf * sp->rc_len;
        inbf++;
        /* --- move the rcd to the buffer --- */
        memmove(*sptr, s_rcd, sp->rc_len);
        sptr++;                 /* point to next array entry */
    }
    else    {               /* null pointer means no more rcds */
        if (inbf)   {       /* any records in the buffer?      */
            qsort(init_sptr, inbf,
                sizeof (char *), comp);
            if (no_seq)     /* if this isn't the only sequence*/
                dumpbuff();  /* dump the buffer to a work file */
            no_seq++;        /* count the sequence             */
        }
        no_seq1 = no_seq;
        if (no_seq > 1)    /* if there is more than 1 sequence */
            prep_merge();  /* prepare for the merge            */
    }
}

/* -------------- Prepare for the merge ----------------- */
static void prep_merge()
{
    int i;
    struct bp *rr;
    unsigned n_bfsz;

    memset(init_sptr, '\0', bspace);
    /* -------- merge buffer size ------ */
```

(continued on page 146)

Listing Three *(Listing continued, text begins on page 127.)*

```
    n_bfsz = bspace - no_seq * sizeof(struct bp);
    /* ------ # rcds/seq in merge buffer ------- */
    rcds_seq = n_bfsz / no_seq / sp->rc_len;
    if (rcds_seq < 2)   {
        /* ---- more sequence blocks than will fit in buffer,
                merge down ---- */
        fp2 = wopen(f2name, 2);     /* open a sort work file */
        while (rcds_seq < 2)    {
            FILE *hd;
            merge();                /* binary merge */
            hd = fp1;               /* swap fds */
            fp1 = fp2;
            fp2 = hd;
            nrcds *= 2;
            /* ------ adjust number of sequences ------ */
            no_seq = (no_seq + 1) / 2;
            n_bfsz = bspace - no_seq * sizeof(struct bp);
            rcds_seq = n_bfsz / no_seq / sp->rc_len;
        }
    }
    bf1 = init_sptr;
    rr = (struct bp *) init_sptr;
    bf1 += no_seq * sizeof(struct bp);
    bf = bf1;

    /* fill the merge buffer with records from all sequences */

    for (i = 0; i < no_seq; i++)    {
        fseek(fp1, (long) i * ((long) nrcds * sp->rc_len),
                    SEEK_SET);
        /* ------ read them all at once ------ */
        fread(bf1, rcds_seq * sp->rc_len, 1, fp1);
        rr->rc = bf1;
        /* --- the last seq has fewer rcds than the rest --- */
        if (i == no_seq-1)  {
            if (totrcd % nrcds > rcds_seq)      {
                rr->rbuf = rcds_seq;
                rr->rdsk = (totrcd % nrcds) - rcds_seq;
            }
            else        {
                rr->rbuf = totrcd % nrcds;
                rr->rdsk = 0;
            }
        }
        else        {
            rr->rbuf = rcds_seq;
            rr->rdsk = nrcds - rcds_seq;
        }
        rr++;
        bf1 += rcds_seq * sp->rc_len;
    }
}
```

```
/* ------- Merge the work file down
    This is a binary merge of records from sequences
    in fp1 into fp2. ------ */
static void merge()
{
    int i;
    int needy, needx;   /* true = need a rcd from (x/y)   */
    int xcnt, ycnt;     /* # rcds left each sequence      */
    int x, y;           /* sequence counters              */
    long adx, ady;      /* sequence record disk addresses */

    /* --- the two sets of sequences are x and y ----- */
    fseek(fp2, 0L, SEEK_SET);
    for (i = 0; i < no_seq; i += 2)     {
        x = y = i;
        y++;
        ycnt =
            y == no_seq ? 0 : y == no_seq - 1 ?
            totrcd % nrcds : nrcds;
        xcnt = y == no_seq ? totrcd % nrcds : nrcds;
        adx = (long) x * (long) nrcds * sp->rc_len;
        ady = adx + (long) nrcds * sp ->rc_len;
        needy = needx = 1;
        while (xcnt || ycnt)    {
            if (needx && xcnt)  {   /* need a rcd from x? */
                fseek(fp1, adx, SEEK_SET);
                adx += (long) sp->rc_len;
                fread(init_sptr, sp->rc_len, 1, fp1);
                needx = 0;
            }
            if (needy && ycnt)  {   /* need a rcd from y? */
                fseek(fp1, ady, SEEK_SET);
                ady += sp->rc_len;
                fread(init_sptr+sp->rc_len, sp->rc_len, 1, fp1);
                needy = 0;
            }
            if (xcnt || ycnt)   {   /* if anything is left */
                /* ---- compare the two sequences --- */
                if (!ycnt || (xcnt &&
                    (comp(&init_sptr, &init_sptr + sp->rc_len))
                        < 0))   {
                    /* ----- record from x is lower ---- */
                    fwrite(init_sptr, sp->rc_len, 1, fp2);
                    --xcnt;
                    needx = 1;
                }
                else if (ycnt)  {  /* record from y is lower */
                    fwrite(init_sptr+sp->rc_len,
                                sp->rc_len, 1, fp2);
                    --ycnt;
                    needy = 1;
                }
            }
        }
    }
}

/* -------- Dump the sort buffer to the work file ---------- */
static void dumpbuff()
{
    int i;

    if (fp1 == NULL)
        fp1 = wopen(fdname, 1);
    sptr = (char **) init_sptr;
    for (i = 0; i < inbf; i++)  {
        fwrite(*(sptr + i), sp->rc_len, 1, fp1);
        *(sptr + i) = 0;
    }
    inbf = 0;
}

/* --------------- Open a sort work file ------------------- */
static FILE *wopen(char *name, int n)
{
    FILE *fp;
    strcpy(name, "sortwork.000");
    name[strlen(name) - 1] += n;
    if ((fp =  fopen(name, "wb+")) == NULL) {
        printf("\nFile error");
        exit(1);
    }
    return fp;
}

/* --------- Function to get sorted records ----------------
This is called to get sorted records after the sort is done.
It returns pointers to each sorted record.
Each call to it returns one record.
When there are no more records, it returns NULL. ------ */

char *sort_op()
{
    int j = 0;
    int nrd, i, k, l;
    struct bp *rr;
    static int rl = 0;
    char *rtn;
    long ad, tr;

    sptr = (char **) init_sptr;
    if (no_seq < 2) {
        /* -- with only 1 sequence, no merge has been done -- */
        if (rl == totrcd)   {
            free(init_sptr);
            fp1 = fp2 = NULL;
            rl = 0;
            return NULL;
        }
        return *(sptr + rl++);
    }
```

```
    rr = (struct bp *) init_sptr;
    for (i = 0; i < no_seq; i++)
        j |= (rr + i)->rbuf | (rr + i)->rdsk;

    /* -- j will be true if any sequence still has records - */
    if (!j)     {
        fclose(fp1);            /* none left */
        remove(fdname);
        if (fp2)    {
            fclose(fp2);
            remove(f2name);
        }
        free(init_sptr);
        fp1 = fp2 = NULL;
        r1 = 0;
        return NULL;
    }
    k = 0;

    /* --- find the sequence in the merge buffer
                    with the lowest record --- */
    for (i = 0; i < no_seq; i++)
        k = ((comp( &(rr + k)->rc, &(rr + i)->rc) < 0) ? k : i);

    /* --- k is an integer sequence number that offsets to the
        sequence with the lowest record ---- */

    (rr + k)->rbuf--;           /* decrement the rcd counter */
    rtn = (rr + k)->rc;         /* set the return pointer */
    (rr + k)->rc += sp->rc_len;
    if ((rr + k)->rbuf == 0)    {
        /* ---- the sequence got empty ---- */
        /* --- so get some more if there are any --- */
        rtn = bf + k * rcds_seq * sp->rc_len;
        memmove(rtn, (rr + k)->rc - sp->rc_len, sp->rc_len);
        (rr + k)->rc = rtn + sp->rc_len;
        if ((rr + k)->rdsk != 0)    {
            l = ((rcds_seq-1) < (rr+k)->rdsk) ?
                rcds_seq-1 : (rr+k)->rdsk;
            nrd = k == no_seq - 1 ? totrcd % nrcds : nrcds;
            tr = (long) ((k * nrcds + (nrd - (rr + k)->rdsk)));
            ad = tr * sp->rc_len;
            fseek(fp1, ad, SEEK_SET);
            fread(rtn + sp->rc_len, l * sp->rc_len, 1, fp1);
            (rr + k)->rbuf = l;
            (rr + k)->rdsk -= l;
        }
        else
            memset((rr + k)->rc, 127, sp->rc_len);
    }
    return rtn;
}
```

```
/* ------- Function to display sort stats -------- */
void sort_stats()
{
    printf("\n\n\nRecord Length = %d",sp->rc_len);
    printf("\n%d records sorted",totrcd);
    printf("\n%d sequence",no_seq1);
    if (no_seq1 != 1)
        putchar('s');
    printf("\n%u characters of sort buffer", bspace);
    printf("\n%d records per buffer\n\n",nrcds1);
}

/* ----- appropriate available memory ----- */
static char *appr_mem(unsigned *h)
{
    char *buff = NULL;

    *h = (unsigned) MOSTMEM + 1024;
    while (buff == NULL && *h > LEASTMEM)   {
        *h -= 1024;
        buff = malloc(*h);
    }
    return buff;
}

/* ------- compare function for sorting, merging -------- */
static int comp(char **a, char **b)
{
    int i, k;

    if (**a == 127 || **b == 127)
        return (int) **a - (int) **b;
    for (i = 0; i < NOFLDS; i++)    {
        if (sp->s_fld[i].f_pos == 0)
            break;
        if ((k = strnicmp((*a)+sp->s_fld[i].f_pos - 1,
                          (*b)+sp->s_fld[i].f_pos - 1,
                          sp->rc_len)) != 0)
            return (sp->s_fld[i].ad == 'd')?-k:k;
    }
    return 0;
}
```

End Listings

Listing One *(Text begins on page 135.)*

```
{-------------------------------------------------}
{                    TIMEDATE                     }
{                                                 }
{ A Time-and-date stamp object for Turbo Pascal 5.5 }
{                                                 }
{                          by Jeff Duntemann      }
{                          Last update 12/23/89   }
{                                                 }
{ NOTE: This unit should be good until December 31, }
{ 2043, when the long integer time/date stamp turns }
{ negative.  HOWEVER, the Zeller's Congruence     }
{ algorithm shown here fails at the end of the 20th }
{ century.  I should be able to figure out the fix  }
{ by then...                                      }
{-------------------------------------------------}

UNIT TimeDate;

INTERFACE

USES DOS;

TYPE
  String9  = STRING[9];
  String20 = STRING[20];
  String50 = STRING[50];

  When =
    OBJECT
      WhenStamp      : LongInt;      { Combined time/date stamp }
      TimeString     : String9;      { i.e., "12:45a"           }
      Hours,Minutes,Seconds : Word;  { Seconds is always even!  }
      DateString     : String20;     { i.e., "06/29/89"         }
      LongDateString : String50;     { i.e., "Thursday, June 29, 1989" }
      Year,Month,Day : Word;
      DayOfWeek      : Integer;      { 0=Sunday, 1=Monday, etc. }
      FUNCTION GetTimeStamp : Word;  { Returns DOS-format time stamp }
      FUNCTION GetDateStamp : Word;  { Returns DOS-format date dtamp }
      PROCEDURE PutNow;
      PROCEDURE PutWhenStamp(NewWhen  : LongInt);
      PROCEDURE PutTimeStamp(NewStamp : Word);
      PROCEDURE PutDateStamp(NewStamp : Word);
      PROCEDURE PutNewDate(NewYear,NewMonth,NewDay : Word);
      PROCEDURE PutNewTime(NewHours,NewMinutes,NewSeconds : Word);
    END;


IMPLEMENTATION

{ Keep in mind that all this stuff is PRIVATE to the unit! }

CONST
  MonthTags : ARRAY [1..12] of String9 =
    ('January','February','March','April','May','June','July',
     'August','September','October','November','December');
  DayTags   : ARRAY [0..6] OF String9 =
    ('Sunday','Monday','Tuesday','Wednesday',
     'Thursday','Friday','Saturday');

TYPE
  WhenUnion =
    RECORD
      TimePart : Word;
      DatePart : Word;
    END;

VAR
  Temp1 : String50;
  Dummy : Word;

{ Some utility routines private to this unit: }

FUNCTION CalcTimeStamp(Hours,Minutes,Seconds : Word) : Word;

BEGIN
  CalcTimeStamp := (Hours SHL 11) OR (Minutes SHL 5) OR (Seconds SHR 1);
END;


FUNCTION CalcDateStamp(Year,Month,Day : Word) : Word;

BEGIN
  CalcDateStamp := ((Year - 1980) SHL 9) OR (Month SHL 5) OR Day;
END;


PROCEDURE CalcTimeString(VAR TimeString : String9;
                         Hours,Minutes,Seconds : Word);

VAR
  Temp1,Temp2 : String9;
  AMPM        : Char;
  I           : Integer;

BEGIN
  I := Hours;
  IF Hours = 0 THEN I := 12;   { "0" hours = 12am }
  IF Hours > 12 THEN I := Hours - 12;
  IF Hours > 11 THEN AMPM := 'p' ELSE AMPM := 'a';
  Str(I:2,Temp1); Str(Minutes,Temp2);
  IF Length(Temp2) < 2 THEN Temp2 := '0' + Temp2;
  TimeString := Temp1 + ':' + Temp2 + AMPM;
END;


PROCEDURE CalcDateString(VAR DateString : String20;
                         Year,Month,Day : Word);
BEGIN
  Str(Month,DateString);
  Str(Day,Temp1);
  DateString := DateString + '/' + Temp1;
  Str(Year,Temp1);
  DateString := DateString + '/' + Copy(Temp1,3,2);
END;


PROCEDURE CalcLongDateString(VAR LongdateString : String50;
                             Year,Month,Date,DayOfWeek : Word);
VAR
  Temp1 : String9;

BEGIN
  LongDateString := DayTags[DayOfWeek] + ', ';
  Str(Date,Temp1);
  LongDateString := LongDateString +
    MonthTags[Month] + ' ' + Temp1 + ', ';
  Str(Year,Temp1);
  LongDateString := LongDateString + Temp1;
END;


{-------------------------------------------------------------------}
{ This calculates a day of the week figure, where 0=Sunday, 1=Monday, }
{ and so on, given the year, month, and day.  The year may be passed  }
{ as either "1989" or "89" but *not* as 1980-relative, or "9".  Also  }
{ note that this particular algorithm turns into a pumpkin in 2000.   }
{ BTW, don't ask me to explain how this crazy thing works.  I haven't }
{ the foggiest notion.  If I ever meet Mr. Zeller, I'll ask him.      }
{-------------------------------------------------------------------}

FUNCTION CalcDayOfWeek(Year,Month,Day : Word) : Integer;

VAR
  Century,Leftovers,Holder : Integer;

BEGIN
  { First test for error conditions on input values: }
  IF (Year < 0)  OR
     (Month < 1) OR (Month > 12) OR
     (Day < 1)   OR (Day > 31) THEN
     CalcDayOfWeek := -1  { Return -1 to indicate an error }
  ELSE
    { Do the Zeller's Congruence calculation: }
    BEGIN
      IF Year < 100 THEN Inc(Year,1900);
      Dec(Month,2);
      IF (Month < 1) OR (Month > 10) THEN
        BEGIN
          Dec(Year,1);
         Inc(Month,12);
        END;
      Century   := Year DIV 100;
      Leftovers := Year MOD 100;
      Holder    := (Trunc(Int(2.6 * Month - 0.2)) + Day +
                    Leftovers + (Leftovers DIV 4) +
                    (Century DIV 4) - Century - Century) MOD 7;
      IF Holder < 0 THEN
        Inc(Holder,7);
      CalcDayOfWeek := Holder;
    END;
END;


{*************************************}
{ Method implementations for type When: }
{*************************************}


{-------------------------------------------------------------------}
{ There will be many times when an individual date or time stamp will }
{ be much more useful than a combined time/date stamp.  These simple  }
{ functions return the appropriate half of the combined long integer  }
{ time/date stamp without incurring any calculation overhead.  It's   }
{ done with a simple value typecast:                                  }
{-------------------------------------------------------------------}

FUNCTION When.GetTimeStamp : Word;

BEGIN
  GetTimeStamp := WhenUnion(WhenStamp).TimePart;
END;


FUNCTION When.GetDateStamp : Word;

BEGIN
  GetDateStamp := WhenUnion(WhenStamp).DatePart;
END;


{-------------------------------------------------------------------}
{ To fill a When record with the current time and date as maintained  }
{ by the system clock, execute this method:                           }
{-------------------------------------------------------------------}

PROCEDURE When.PutNow;

BEGIN
  { Get current clock time.  Note that we ignore hundredths figure: }
  GetTime(Hours,Minutes,Seconds,Dummy);
  { Calculate a new time stamp and update object fields: }
  PutTimeStamp(CalcTimeStamp(Hours,Minutes,Seconds));
  GetDate(Year,Month,Day,Dummy); { Get current clock date }
  { Calculate a new date stamp and update object fields: }
  PutDateStamp(CalcDateStamp(Year,Month,Day));
END;


{-------------------------------------------------------------------}
{ This method allows us to apply a whole long integer time/date stamp }
{ such as that returned by the DOS unit's GetFTime procedure to the   }
```

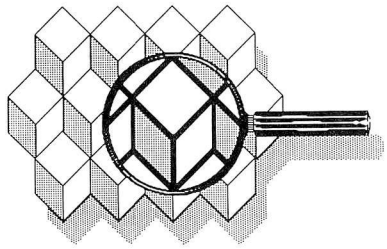
```
{ When object.  The object divides the stamp into time and date       }
{ portions and recalculates all other fields in the object.           }
{---------------------------------------------------------------------}

PROCEDURE When.PutWhenStamp(NewWhen  : LongInt);

BEGIN
  WhenStamp := NewWhen;
  { We've actually updated the stamp proper, but we use the two }
  { "put" routines for time and date to generate the individual }
  { field and string representation forms of the time and date. }
  { I know that the "put" routines also update the long integer }
  { stamp, but while unnecessary it does no harm.               }
  PutTimeStamp(WhenUnion(WhenStamp).TimePart);
  PutDateStamp(WhenUnion(WhenStamp).DatePart);
END;


{---------------------------------------------------------------------}
{ We can choose to update only the time stamp, and the object will    }
{ recalculate only its time-related fields.                           }
{---------------------------------------------------------------------}

PROCEDURE When.PutTimeStamp(NewStamp : Word);

BEGIN
  WhenUnion(WhenStamp).TimePart := NewStamp;
  { The time stamp is actually a bitfield, and all this shifting left }
  { and right is just extracting the individual fields from the stamp:}
  Hours := NewStamp SHR 11;
  Minutes := (NewStamp SHR 5) AND $003F;
  Seconds := (NewStamp SHL 1) AND $001F;
  { Derive a string version of the time: }
  CalcTimeString(TimeString,Hours,Minutes,Seconds);
END;


{---------------------------------------------------------------------}
{ Or, we can choose to update only the date stamp, and the object     }
{ will then recalculate only its date-related fields.                 }
{---------------------------------------------------------------------}

PROCEDURE When.PutDateStamp(NewStamp : Word);

BEGIN
  WhenUnion(WhenStamp).DatePart := NewStamp;
  { Again, the date stamp is a bit field and we shift the values out  }
  { of it: }
  Year := (NewStamp SHR 9) + 1980;
  Month := (NewStamp SHR 5) AND $000F;
  Day := NewStamp AND $001F;
  { Calculate the day of the week value using Zeller's Congruence:    }
  DayOfWeek := CalcDayOfWeek(Year,Month,Day);
  { Calculate the short string version of the date; as in "06/29/89": }
  CalcDateString(DateString,Year,Month,Day);
  { Calculate a long version, as in "Thursday, June 29, 1989": }
  CalcLongDateString(LongdateString,Year,Month,Day,DayOfWeek);
END;


PROCEDURE When.PutNewDate(NewYear,NewMonth,NewDay : Word);

BEGIN
  { The "boss" field is the date stamp.  Everything else is figured }
  { from the stamp, so first generate a new date stamp, and then    }
  { (odd as it may seem) regenerate everything else, *including*    }
  { the Year, Month, and Day fields: }
  PutDateStamp(CalcDateStamp(NewYear,NewMonth,NewDay));
  { Calculate the short string version of the date; as in "06/29/89": }
  CalcDateString(DateString,Year,Month,Day);
  { Calculate a long version, as in "Thursday, June 29, 1989": }
  CalcLongDateString(LongdateString,Year,Month,Day,DayOfWeek);
END;


PROCEDURE When.PutNewTime(NewHours,NewMinutes,NewSeconds : Word);

BEGIN
  { The "boss" field is the time stamp.  Everything else is figured }
  { from the stamp, so first generate a new time stamp, and then    }
  { (odd as it may seem) regenerate everything else, *including*    }
  { the Hours, Minutes, and Seconds fields: }
  PutTimeStamp(CalcTimeStamp(NewHours,NewMinutes,NewSeconds));
  { Derive the string version of the time: }
  CalcTimeString(TimeString,Hours,Minutes,Seconds);
END;
```

End Listing One

Listing Two

```
PROGRAM TimeTest;

USES Crt,TimeDate;

VAR
  Now : When;

BEGIN
  Write('At the tone, it will be exactly ');
  Delay(1000);
  Now.PutNow;
  Sound(1000); Delay(100); NoSound;
  WITH Now DO Writeln(TimeString,'m on ',LongDateString,'.');
  Readln
END.
```

End Listings



Actor 2.0, an object-oriented development environment for Microsoft Windows applications, has been announced by **The Whitewater Group**. Version 2.0 boasts an automatic object swapping system that swaps static objects and code out to disk, which allows developers to break the 640K barrier of MS-DOS and create applications that are larger than 1 Mbyte in size. It does this by means of an LRU (least-recently used) algorithm, which ages unused objects and sends the old ones out to disk.

Actor 2.0 also has additional object-oriented features, new commands, and improved support for C. Whitewater spokesman Zack Urlocker told *DDJ* that "programmers wanted increased compatibility with C so we've added the ability to pass C structures and combine primitives in C and assembly language. This means you can get directly into the low-level structure of objects." 2.0 runs on any PC or PS/2 (or compatibles) under Microsoft Windows, and requires 640K of memory, a hard disk, a graphics card of any resolution, a mouse, and Microsoft Windows 2.x or later. The cost is $695, but registered users of previous versions can upgrade for $149. Reader service no. 20.
The Whitewater Group
600 Davis St.
Evanston, IL 60201
708-328-3800

Version 2.00 of Top Speed Modula-2 has been announced by **JPI**. This version works with the multilanguage development recently released with TopSpeed C, which automatically selects the appropriate compiler for the source file it needs to compile, and includes nine editing windows, 500K per file, and a hypertext help system. New compiler options allow interfacing of C libraries and functions with Modula-2 programs, and vice versa. The optimizing code generator is featured in the new family of TopSpeed languages (including a TopSpeed Pascal, which has also been announced), common to all the languages.

TopSpeed Modula-2, Version 2.00, has multiple memory model support, and new keywords and syntax extensions allow object-oriented programming. A new keyword CLASS allows a RECORD-like structure to include PROCEDURE declarations, hide data, specify inheritance and VIRTUAL procedures. And syntax extensions allow the name of a variable of a CLASS type to qualify the name of a procedure.

Included now in all TopSpeed compilers is the Visual Interactive Debugger (VID), which is a multilanguage source-level debugger that features on-screen display of all breakpoint traps, single-step operation, and full source code inspection. TopSpeed Modula-2 is available for DOS ($395 for extended edition) and OS/2 ($495 for extended edition). Reader service no. 33.
Jensen & Partners International
1101 San Antonio Rd., Ste. 301
Mountain View, CA 94043
415-967-3200

A software development tool that gives the 80386 processor the same level of protection available in protected operating systems has been announced by **Nu-Mega**. Bounds-Checker automatically detects out-of-bounds accesses by an application program, using the symbolic information created by the Microsoft C 5.X and 6.0 compilers to show you the exact source line causing the out-of-bounds access. Nu-Mega claims that Bounds-Checker cuts steps in the development process and eliminates the need to debug, and that it prevents serious side effects of subtle overwrites that may not be particularly dangerous until the program is in the field.

The Bounds-Checker provides real-time memory protection, differentiates between code and data, protects program code and all memory outside your program, and prevents the system software from corrupting your program — it can determine if a TSR or other program is trouncing your program. Sells for $249. Reader service no. 35.
Nu-Mega Technologies
P.O. Box 7607
Nashua, NH 03060-7607
603-888-2386

Borland International has announced Version 2.0 of its Turbo Debugger, which includes a toolkit that features the new Turbo Profiler. The profiler measures where in your program time is spent, how many times a line is executed, how many times a routine is called and by what, and which files are accessed most often and for how long. It also tracks the use of resources such as processor time, disk access, keyboard input, printer output, and interrupt activity.

The Turbo Profiler graphically displays where your program is spending its time, telling you which parts of the program are used most often and may need optimization or rewriting, and which parts are used so little that you needn't bother tightening them. An optimizing compiler generates code for the program you give it.

Multiple overlapping windows, icons, mouse support, and context-sensitive on-line help are included in the user interface. The Profiler works with Turbo Pascal 5.0, Turbo C 2.0, and Turbo Assembler 1.0, and any later versions of these compilers. Also supports CodeView and .MAP file debug formats. For IBM PCs and compatibles operating PC-DOS (MS-DOS) 2.0 or later. Requires 384K (256K for Turbo Assembler). The toolkit retails for $149.95. Reader service no. 34.
Borland International
1800 Green Hills Rd.
Scotts Valley, CA 95066-0001
408-438-8400

Version 4.0 of the LALR compiler construction toolkit can now be purchased from **LALR Research**. The toolkit includes the parser generator, LALR, and a scanner generator, DFA, as well as various source code modules, such as a main program, a screener, parser skeleton, and scanner skeleton. *DDJ* spoke with the developer, Paul Mann, who said that the LALR compiler is "an advanced compiler construction tool that goes beyond YACC. It features extended BNF notation, automatic creation of abstract syntax trees, and a high-speed scanner generator. It's well suited for developing compilers, translators, and interpreters for computer languages."

A BNF grammar describes the statements of the language as input to the LALR, and describes the symbols as input to the DFA. The LALR parser generator provides automatic error recovery, and handles large grammars such as Fortran and Cobol. The company claims that the DFA scanner generator produces high-speed deterministic finite automatons that run about four times faster than LEX scanners. The source code output is compatible with Turbo C, Microsoft C, Watcom C, among others. The price is $495. Reader service no. 26.
LALR Research
1892 Burnt Mill Rd.
Tustin, CA 92680
714-832-2274

(continued on page 157)

(continued from page 152)
BrainMaker v2.0, a system for designing, building, training, testing, and running neural networks, is now available from **California Scientific Software**. The company claims that version 2.0 is a major enhancement, and has such features as the NetMaker, which is a network generation and data manipulation program with spread-sheet style data display and the ability to perform arithmetic operations on data. In addition, BrainMaker now reads data from Lotus, dBase, Quattro, and Excel files; automatic numeric translation allows the display of large numbers; graphics post-processing of network results is supported; and it includes training and running algorithms that are as fast as 500,000 neural connections per second. Mark Lawrence, CSS president, told *DDJ* that "with Release 1, we knew we had a neat technology, and that our users would have to tell us what it was for. They told us it was mostly for financial forecasting, so now we've gone back and written it like it should have been."

This upgrade includes the *Introduction to Neural Networks*, an overview of the history and research of neural networks, and a user's guide and reference manual. The cost is $195, $95 for registered users. Requires a PC, PS/2, or compatible. Reader service no. 21.

California Scientific Software
160 E. Montecito Ave., #E
Sierra Madre, CA 91024
818-355-1094

Macintosh Allegro Common Lisp (CL), v. 1.3, an extended implementation of Common Lisp, is being shipped by **Apple Computer**. It supports all the features described in the Guy Steele text *Common Lisp: The Language*. The Macintosh Allegro CL can be used to develop stand-alone Macintosh applications and to port applications developed on other machines.

User interface components can be modified both interactively and under program control. Events are caught and dispatched by the Lisp run-time kernel. Windows are accessible as high-level objects, and can be created and closed with simple Lisp functions. Menus and dialog boxes are implemented as objects, as well. Fred, an integrated programmable editor, combines the capabilities of Emacs with the multiple-window, mouse-based editing style of the Mac. The Stand-Alone Application Generator produces ready-to-use Mac applications, which require a "nominal fee" license to distribute. System requirements include any Mac except the 512K, a second 800K disk drive, and

Mac System Software, Version 6.0. It sells for $495. Reader service no. 28.
Apple Computer, Inc.
20525 Mariani Ave.
Cupertino, CA 95014
408-996-1010

MS-DOS Kermit, Version 3.0, is now available from the **Columbia University Center for Computing Activities**. The new features include transfer of text files in international character sets via a new Kermit protocol extension; emulation of the DEC VT320 terminal, including soft function keys and support for a wide variety of international character sets in any of the five standard PC code pages; and sliding window packet protocols for improved file transfer performance over public data networks and long distance satellite connections.

Version 3.0 also has expanded support for local area networks, and enhanced Tektronix graphics terminal emulation with VT340 extensions. Graphics screens may be saved in TIFF 5.0 for importation into such applications as WordPerfect, Pagemaker, and Ventura Publisher. This version was prepared by Joe R. Doupnik of Utah State University in cooperation with Columbia University. Reader service no. 29.
Kermit Distribution, Columbia Univ.
Center for Computing Activities
612 W. 115th St.
New York, NY 10025
212-854-3703

Stony Brook Professional Modula-2, Version 2.0, is now available from **Stony Brook Software**. This compiler package includes the QuickMod compiler and an optimizing compiler; development support for DOS, OS/2, and Microsoft Windows; the ability to interface with libraries written in C or other languages; the M16 debugger; an extensive run-time library; built-in multitasking; and the Stony Brook linker.

The Stony Brook environment gives you control over source file placement, keystroke macros, the ability to perform DOS commands, a cross-reference of module dependencies, interface to the symbolic debugger, and the ability to assemble foreign language modules. QuickMod complies with Wirth's definition of Modula-2, and supports symbol scoping and forward reference capabilities in one pass. Added are structured constants, array substrings, conditional compilation, type coercions, and set types containing up to 65,536 bits. The whole package retails for $295; the source code for the run-time library costs $150. Reader service no. 24.
Stony Brook Software
187 E. Wilbur R., Ste. 9
Thousand Oaks, CA 91360
800-624-7487 (US)
805-496-5837 (Calif. and International)

Books of Interest

NeuralSource, The Bibliographic Guide to Artificial Neural Networks, by Phillip Wasserman and Roberta Oetzel, is available from **Van Nostrand Reinhold**. This bibliography purports to be the most extensive collection of research information on neural nets. Periodicals, private reports, and books are included. Sells for $64.95. ISBN 0-442-23776-6.

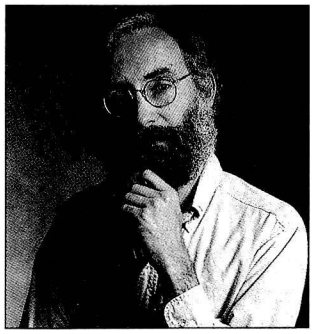
Also by Wasserman is *Neural Computing, Theory and Practice*. This is an introduction to artificial neural networks for the nonspecialist. Assumes no math background beyond an undergraduate scientific education. Uses a step-by-step algorithmic approach to present commonly used network paradigms. $36.95, ISBN 0-442-20743-3. Reader service no. 30.
Van Nostrand Reinhold
P.O. Box 668
Florence, KY 41022-0668
606-525-6600

Elements of Functional Programming by Chris Reade has been published by **Addison-Wesley**. Covers the concepts and techniques used in modern functional programming languages, as well as support for abstraction, programming with lists, new types, abstract data types and modules, lazy and eager evaluation, and implementation techniques. Hardback edition costs $37.75, ISBN 0-201-12915-9. Reader service no. 31.
Addison-Wesley
Reading, MA 01867
617-944-3700

The *COSMIC Software Catalog 1990 Edition* is available from the **University of Georgia**. It is a comprehensive listing of program abstracts describing all available NASA computer programs. You can purchase it in book form ($25), on microcomputer diskette ($30), on magnetic tape ($50), or on microfiche ($10). The catalog cross-indexes over 1200 computer programs, in areas such as aerodynamics, reliability, composites, heat transfer, artificial intelligence, and structural analysis. Reader service. no. 32.
COSMIC
University of Georgia
382 E. Broad St.
Athens, GA 30602
404-542-4807

(continued on page 125)

Junk Customers



2/11/90: Investment banking firm Drexel Burnham Lambert, home of the junk bond phenomenon, informs its employees that it is filing for bankruptcy.
2/12/90: My cousin Corbett launches his program for software developers who can't afford the skyrocketing costs of software marketing: Junk Customers.

Corbett was concerned about the software developer who can't afford to run large ads in the major computer magazines and can't afford to rent lists of prospects. Some magazines and some lists will result in more responses and purchases than others, of course, and it was while trying to come up with a new measure of the value of these sources that Corbett hit on the secret, which he calls "Junk Customers."

He took his inspiration from Mike Milken, the Drexel Burnham Lambert employee who made such a splash with junk bonds. Milken noticed that there were a lot of companies that had to go to the bank when they wanted money. This was bad, he realized. The companies tried issuing bonds to get investors to put up money, but investment firms such as Drexel et al. steered investors away from these bonds, labeling them "low value." This meant that there was a higher probability that the companies would fail to pay up — go out of business or whatever — than was the case with so-called high-quality bonds. The companies tried to make their bonds appealing by increasing the yield —what you get back for your investment — and Mike Milken saw this as a good deal. He began helping his clients to buy a broad selection of these high-yield, low-value bonds, starting what became, at its peak, a $200 billion market.

Corbett has come up with a similar plan for software marketing. (The plan is completely general, but his loyalty is to the software development community. He wants you to have it.) He defines the yield of a source of prospects — a list of names or a page of advertising — as the inverse of the CPM (ad sales jargon for "cost per thousand"). Yield is how many names you get for a buck. He defines the value of a source as how well the source will pull — how likely each name is to result in a sale. The trick, as with junk bonds, is to develop a varied portfolio of high-yield, low-value sources.

Identifying a truly low-value source is tricky. It can't just be a source that is ill-suited to your needs; such a source might be able to get a lot of money from someone else. To ensure that the yield can be made high enough, this must be a source ill-suited to anyone's needs, a publication or list poorly suited to any commercial advertising or name rental purpose. Then there is another problem in dealing with low-quality sources: You'll need a lot of them. The low quality translates into few responses from any one source, and the overhead of dealing with hundreds of such companies can easily eat up any gains.

Corbett thinks he has found the single correct answer to the Junk Customers challenge, and is generously allowing me to pass it on to you: Church newsletters. Every community has a church, every church has a newsletter, and every church belongs to some large national or international organization capable of serving as a central clearing house for ad sales or list rental. The nonprofit status of churches and their general, noncommercial slant makes a church newsletter an exceptionally low-value source, Corbett maintains.

He sees an intriguing wrinkle to the idea of church newsletter subscribers as prospects for software sales. Current wisdom says that you should look for software prospects among owners of computers. The church newsletter subscribers will include many who do not own a computer, apparently nonprospects almost by definition. But any good marketer knows to mistrust such self-fulfilling predictions, and to ask the positive question, why would this person want my product? In this case, the answer is surprisingly obvious. The industry has been doing it backwards!

Consider: It is much easier to ease a potential customer into a new product category with a small purchase than with a large one. One of the reasons many people cite for not buying a PC is that they don't know how to justify spending over a thousand dollars. So they buy a Nintendo instead. These people could be buying your CAD package.

Consider: Anyone who has ever thought about buying a PC has heard the advice, "Decide what software you want to run, and then buy the PC that runs that software." You've probably given that advice, but did you listen to what you were really saying?

Consider this pitch: For less than the cost of a Nintendo, you can own the most powerful CAD package in the known universe. Now you, too, can design microprocessor circuits, draft plans for a new house on the coast, develop a new art form, make your own clothes. Required Silicon Graphics IRIS workstation must be purchased separately.

Remember, you read it here first.

Michael Swaine

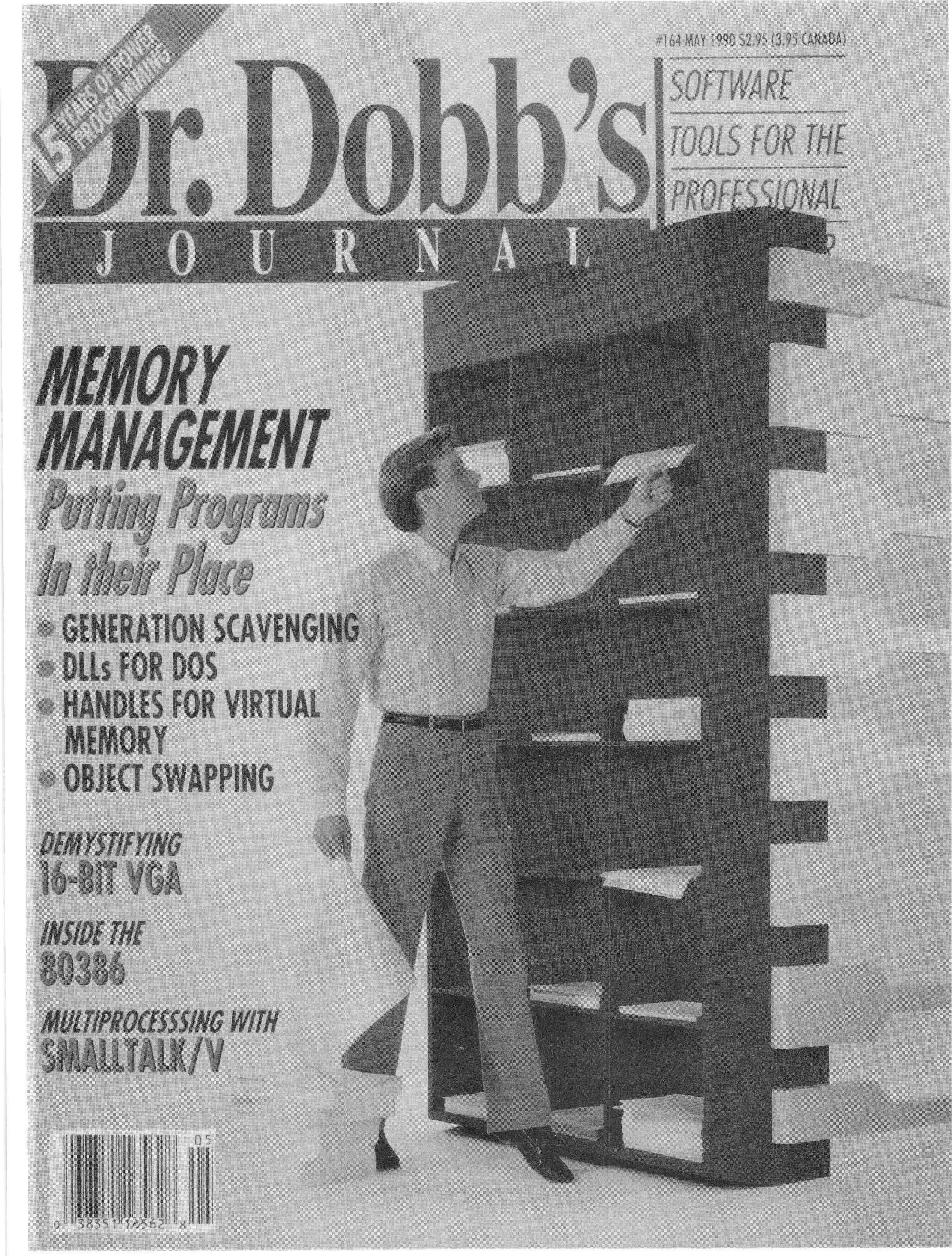
Michael Swaine
editor-at-large

#164 MAY 1990 $2.95 (3.95 CANADA)

15 YEARS OF POWER PROGRAMMING

Dr. Dobb's JOURNAL

SOFTWARE TOOLS FOR THE PROFESSIONAL



MEMORY MANAGEMENT

Putting Programs In their Place

- GENERATION SCAVENGING
- DLLs FOR DOS
- HANDLES FOR VIRTUAL MEMORY
- OBJECT SWAPPING

DEMYSTIFYING 16-BIT VGA

INSIDE THE 80386

MULTIPROCESSSING WITH SMALLTALK/V

0 38351 16562 8 05

CONTENTS

MAY 1990
VOLUME 15, ISSUE 5

FEATURES

GENERATION SCAVENGING 16
by Frank Jackson
The generation scavenging algorithm is an efficient, portable garbage collector that does not require special hardware support.

DYNAMIC LINK LIBRARIES FOR DOS 30
by Gary Syck
DLLs provide an easy way to run large programs in small memory spaces and Gary shows how you can add DLL facilities to DOS programs.

GETTING A HANDLE ON VIRTUAL MEMORY 40
by Walter Bright
"Handle pointers" let you extend available memory space by dynamically allocating data. Walter discusses handles and how you can take advantage of them.

OBJECT SWAPPING 48
by Jan Bottorff and Jim Bolland
For object-oriented environments, "object-swapping" picks up where virtual memory systems leave off.

A MEMORY CONTROLLER 58
by Robert A. Moeser
Rob presents a set of memory management routines that can be used as extensions to your library's *malloc* and *free* routines.

DEMYSTIFYING 16-BIT VGA 70
by Michael Abrash
Michael clears up misconceptions about 16-bit VGA and explains why programmers need not treat 8- and 16-bit VGAs differently.

EXAMINING ROOM

MULTIPROCESSING WITH SMALLTALK/V 82
by Kenneth E. Ayers
Find out what's in store for Ken as he adds multiprocessing capabilities to Smalltalk/V, using the CX Multiprocessing Kit to build a simulated supermarket.

ENCAPSULATING C++ BOOKS 150
by Andrew Schulman
Books on C++ are cropping up all over the place and Andrew has read a lot of them. Here are a few of his favorites.

PROGRAMMER'S WORKBENCH

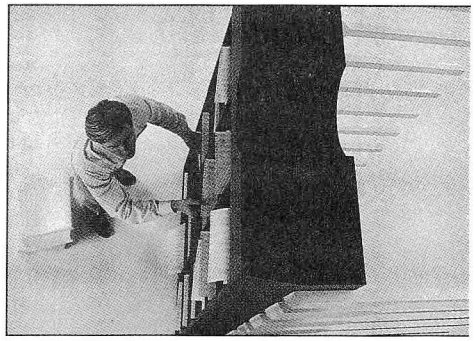
ACCESSING HARDWARE FROM 80386 PROTECTED MODE: PART I 92
by Stephen Fried
Stephen kicks off a two-part discussion of the 80386 by examining topics such as tiling, the huge model, and the use of FAR pointers to address up to 64 terabytes of memory.

COLUMNS

PROGRAMMING PARADIGMS 123
by Michael Swaine
Michael copes with the chaos of complex systems.

C PROGRAMMING 129
by Al Stevens
While many programmers are wondering about moving from C to C++, Al takes the curious step of moving from C++ to C.

STRUCTURED PROGRAMMING 141
by Jeff Duntemann
Jeff poses some sharp questions about cutting edges, explores object hierarchies, and examines the Object Professional Library from Turbo Power Software.



DEPARTMENTS

EDITORIAL 6
by Jonathan Erickson

LETTERS 8
by you

SWAINE'S FLAMES 160
by Michael Swaine

PROGRAMMER'S SERVICES

ADVERTISER INDEX 152
where to go for more information on products

OF INTEREST 153
compiled *by Janna Custer*

PROGRAMMER'S MARKETPLACE 154
classified ads

NEXT ISSUE

Hot-links can make for sizzling software, and in June we'll be tackling hands-on hypertext problems, including the tools for building your own hypertext help system. We'll also continue our examination of C++ and go deeper into the 80386's protected mode.

A Little Help From Our Friends

Depending on where you live, you've probably: a.) just finished shoveling snow for the last time this winter, or b.) mowed your lawn for the first time this summer. (Or maybe, c.) you live in a maintenance-free condo in Florida and don't have to worry about shovels or lawn mowers.) In any case, January 1991 is likely the farthest thing from your mind. For us, magazine seasons being what they are, 1991 is peeking over the masthead and we want your help in zeroing in on our 1991 Editorial Calendar. What we're looking for is a list of general topics you'd like covered in next year's *DDJ*.

We're trying to make it easy on you. If you'll turn to page 16 (home of Frank Jackson's article "Generation Scavenging"), you'll find a tear-out card with a list of possible topics. We'd like you to check-off or rank in order of preference (1 being most preferred) those topics of greatest interest. If the topic you're particularly interested in isn't on the list, use the available space for write-in candidates. And, as a matter of curiosity (and because we had room on the card), tell us which article *DDJ* published over the past few months you liked the best.

Be sure to include your name, address, and phone number, tear out the card, and drop it in the mail. It's already addressed and stamped.

We're also trying to make it worth your while. To provide some incentive for your mailing in the card, we'll have a random drawing around the end of June. If your card is drawn, you'll get your choice of whatever developer's tools are available on the following list. The first person selected gets to pick from the list, then the second person, and so on, right on down the roster. The companies listed below have generously donated their software to help us out, and by the time the drawing actually takes place there may very well be more tools available.

Actor 2.0 — Whitewater Group

Bounds-Checker — Nu-Mega Technologies

dbVista for DOS — Raima Corp.

DDJ *Bound Volume Set* — M&T Books

QRAM and MANIFEST — Quarterdeck Office Systems

Formation — Aspen Scientific

Greenleaf ComLib — Greenleaf Inc.

HCR/C++ — HCR Corp.

Instant-C — Rational Systems Inc.

Microsoft C — Microsoft Corp.

Microsoft Basic — Microsoft Corp.

PCX Toolkit — Genus Microprogramming

Smalltalk V/Mac — Digitalk Inc.

Smalltalk V/286 — Digitalk Inc.

Stony Brook Modula-2 — Stony Brook Software

Think C 4.0 — Symantec Corp.

Think Pascal 3.0 — Symantec Corp.

TopSpeed C or Modula-2 — Jensen & Partners Intl.

Turbo C — Borland Intl.

TurboPower Library — TurboPower

Watcom C or Fortran — Watcom Corp.

Zortech C++ — Zortech Corp.

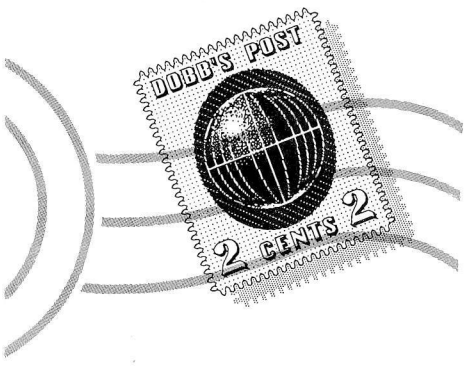
After the drawing, we'll publish a complete list of all winners and the software you selected. You're responsible for any taxes or duties. You don't have to purchase any product to enter the drawing, nor do you have to be a *DDJ* subscriber. You do have to fill out the card, however. (M&T Publishing and Computer Metrics employees aren't eligible.) We'll also publish next year's Editorial Calendar and give you plenty of time to start thinking about and writing articles.

I'm happy to have the opportunity to share these tools with you and am grateful to the vendors who helped out. More than that, however, I'm looking forward to seeing what topics are important to you.

Yes, we're still horsing around . . . It's now official. We made Dr. Dobbs, the quarter horse I wrote about here last month, our honorary mascot after he won his first race. (Note that he's no longer classified as a "nag.") Earlier this racing season, the good Doctor had three photo finishes, all ending on the short side of the wire. Last Saturday's race, however, shaped up as yet another photo finish, but this time with Dr. Dobbs out in front by a nose. The crowd, if not the *DDJ* staff, went wild.



Jonathan Erickson
editor-in-chief



C++ and the 386

Dear *DDJ*,

As one of what I would imagine is a very small number of Intek C++ users, I read with interest the exchange between Mac Cutchins of Intek and Al Stevens in your March 1990 "Letters" section. When I first received the Intek package I had many of the same problems that Al did, including a not so amusing little bug that resulted in the compiler only working every second time. Several of their header files would not compile due to typos and bugs of one sort or another. Intek technical support was always polite but rarely helpful and I sometimes got the impression that I was the only user they had actually suckered into buying the product. To be fair they did supply an upgrade when I complained that the old version of the PharLap binder they had used prevented its working correctly with the VCPI standard and hence precluded the use of QEMM and DesqView. I was also notified (by telephone no less) of their upgrade to 2.0 and it was reasonably priced and delivered promptly.

I was eventually able to contrive the necessary patches, batch files, and bug fixes so that it would reliably compile things from my Brief editor and I could pretend I was working with a real development tool. Still one might ask why bother with it when there are other alternatives.

First and foremost because Intek C++ is the only product which will support the MetaWare HiC or Watcom compilers, thus it is the only way to produce code for 386 protected mode programs running under DOS extenders. Also, as Mac Cutchins points out, Intek's use of 386 protected mode means you are not concerned with running out of memory when compiling large source modules. The large and numerous header files that C++ encourages can quickly exhaust the memory of real mode compilers such as Zortech. Many of my large library modules would have to be split up and might still give problems compiling under Zortech. Finally and unexpectedly, the translator itself is quite robust once it is running. It correctly compiled code segments where Zortech 1.2 version gave spurious errors. The 2.0 version of Zortech seemed more robust in my limited testing of it, but could not compile many of my files due to the memory limitation problem.

I am the only one in our shop using C++ at the moment, but that will change in the near future and I dread having to invest in more copies of the Intek product. With each new issue of *DDJ* I carefully scan all the adds and announcements for a Turbo C++ 386 or something similar. The OS/2 version of Zortech is tempting, but I need to use too many PharLap programs to make that feasible just yet. When all is said and done Intek has the singular advantage of being the only product available under DOS for creating really large C++ applications. If something else is available I would love to know about it.

Craig Morris
Calgary, Alberta, Canada

DDJ *responds: Thanks for your insights, Craig. Just within the last few days,* DDJ *contributing editor Andrew Schulman started an in-depth look of C++ implementations for the 386, beginning with Intek C++ and MicroWay's NDP C++. We're looking forward to sharing his findings sometime in the near future.*

Trick Trade-offs

Dear *DDJ*,

This message is in regards to Tim Paterson's article "Assembly Language Tricks of the Trade" in the March 1990 *DDJ*.

I've always enjoyed reading articles about the tricks and magic that other programmers use. If we assume some things, though, we can do your Binary-To-ASCII Conversion one better.

If we assume that the Carry and Auxiliary Carry are clear, then a binary value in the range 00-0F in AL can be converted to ASCII by:

```
daa             ; 00-09, 10-15
add   Al0F0H    ; F0-F9 NC, 00-05 CY
adc   al,040h   ; 30-39, 41-46 ('0'-'9',
                                'A'-'F')
```

Since we usually want to convert a BYTE to two ASCII characters, this is usually preceded by masking and/or shifting some other value. These operations will clear the Carry and Auxiliary Carry, so everything's OK.

Yet another trick: You mention using the AAM and AAD instructions for Binary/Decimal Conversion. There is an undocumented "extension" to these instructions, which is often useful. The opcodes for AAD and AAM are:

```
AAD     =     D5 0A
AAM     =     D4 0A
```

If the 0As look a little suspicious, it's because they are the divisors used in the conversion. The instruction sequence D4 10 is equivalent to separating the byte in AL into its upper/lower nibbles and placing the upper nibble into the lower nibble of AH, leaving just the lower nibble in AL. This also happens to clear the Carry and Auxiliary Carry flags. Sooooo . . . used in conjunction with the Binary-to-ASCII Conversion code above will result in an extremely compact, brutally fast Byte-to-Two-ASCII-Digits Conversion. Neat, eh?

Keith Moore
Fort Worth, Texas

Tim responds: I am aware of the tricks Keith mentions. However, both rely on undocumented features of the 8086 family, which is a very dangerous practice.

The only instructions which are documented to affect the Auxiliary Carry (AC) flag in a specific way are arithmetic instructions (not including shifts). Masking and shifting instructions are documented as leaving the AC flag undefined. Thus it is very unlikely that the state of the AC flag will be known when Keith's instruction is executed, and the method could easily fail.

Testing with a debugger may leave the impression that masking, for example, leaves the AC flag clear. However, did you check this on an 8088, 8086, 286, or 386? What about the 33-MHz 386, which uses a different mask set than the slower versions? Are you sure the 386SX, 486, and 586 (which no one has seen yet) all work that way?

The same thing can be said for using variants of the AAM and AAD instructions to multiply or divide by something other than ten. Eleven years ago I discovered that the 8086 used the second byte of those instructions as an immediate value. But does a 486? If it does, then the 486 has a bug — it should perform an invalid opcode trap if the second byte is not 0AH. Or else Intel needs to document that it works.

There are too many different processors in the family — and too many different manufacturers — to consider using undocumented features. Let's all play by the rules.

(continued on page 12)

(continued from page 8)

But Basic Already Does That . . .

Dear *DDJ*,

No one is a bigger fan of Jeff Duntemann than I, but he completely missed the boat in his Modula-2 discussion (*DDJ*, February 1990). As Jeff went over the list of omissions in both Pascal and Modula-2, I kept saying to myself, "But QuickBASIC already does that." In my opinion, Microsoft QuickBASIC overcomes all of the shortcomings of both Pascal and Modula-2, with a language that is both fully structured and incredibly easy to use.

For example, Jeff laments Pascal's inability to view a list of procedures, and praises that feature in Modula-2. But QuickBASIC has had a "View Subs" menu for years. He then compares Pascal's ability to use a varying number and type of parameters for built-in statements, as opposed to Modula with its separate *WriteString*, *ReadInt*, and so forth. Again, QuickBASIC (and even interpreted BASIC!) has always had that capability. Worse still, procedures in either language cannot accept a truly "open ended" array. And again, QuickBASIC lets you pass any array — with any number of dimensions and any range of upper and lower bounds — to any subroutine. How else could one write a usable sort routine?!

I won't belabor the remaining list of advantages that QuickBASIC has over the "Wirth" languages. No, I won't dwell on QuickBASIC's many data types, automatic support for a coprocessor, TRUE dynamic strings, world-coordinate graphics, or its ability to manage an entire project without requiring all of the files to be in the same directory. (Yeah, that's a good one — multiple copies of your debugged subroutines scattered all over a disk.) And I won't even belabor QuickBASIC's outstanding support for fully interrupt-drive communications. Where Jeff is bragging about a 100-line Comm program he wrote in an hour using Modula-2, I maintain the same could be done in, say, 20 lines in ten minutes using QuickBASIC.

Indeed, if any language is the rightful successor to "king" Turbo Pascal, surely it is QuickBASIC.

Ethan Winer
Stamford, Connecticut

Editor's note: Ethan is president of Crescent Software, developers of QuickBASIC add-on tools.

Forth-Coming

Dear *DDJ*,

I read Martin Tracy's article, "Zen Forth," with great interest (*DDJ*, January 1990). As a Forth programmer myself, I'm interested in Forth systems and applications. I even wrote a Forth system for sale (CorrectForth — I published it as a product of Correct Software, Inc.). I have a number of comments on the implementation and what looks like bugs in the source code.

First, you could put the address of colon into the register DI. Then colon looks like this:

```
LABEL COLON BP SPXGHGSI PUSH
      BPSP X CHG SI POP NEXT C;
```

(the CFA code or a colon definition is DI CALL). The result is a system about 290 faster than a JMP colon and numerous changes to the source code (string operators, FIND, etc.). The changes are minor and would involve saving and restoring DI. Another change would be to use register ES to point to RAM, thus increasing the amount of code space and data space available. Only string operations would be affected and would involve saving and restoring ES. Then, too, you could describe another register to hold the nest to top of stack value. This speeds up the system by 10 percent since lots of Forth words use 2 parameters. The system as published in *DDJ* runs the Sieve of Erastothenes benchmarks in 46 seconds, but the new improved system in 45 second. Time counts in real-time applications!

The source code bugs are as follows:

Screen	Page	Bug(s)
13	98	use of TRUE (a code defined word) in = <, U<
14	98	same as above, only for 0=, 0<
37	102	use of SPO in depth

The reason I'd call them bugs is that I don't think the metacompiler Martin was using would execute works defined in the metacompiler's target dictionary. If it did, I'd think twice before I'd use such a "feature" — I would cross compile into a processor that might not execute host code . . . !

Overall, this system sings pretty good. I counted on that — Mr. Tracy's been in the Forth community much longer than I have. The choice of a DCT (direct threaded code) implementation of Forth is the best in my opinion since it has the best tradeoff of size vs. speed. If you want speed and don't care about size, go for STC (subroutine threaded code) (like Small C did). If you want really tight code (say you only have 4K of ROM), go for TTC (token threaded code). If you want speed and just have to have small size, go for DTC. The high-level words run at an acceptable speed and providing you chose the proper words as going into assembler (CODE definitions for the knowledgeable), you'll get screaming speed at little cost.

Russell McCale
New York, New York

Martin responds: Thank you for your interest in ZEN Forth. I am writing this letter to answer some of the many questions I have received.

ZEN is a personal dialect I have been developing and porting for several years. Most recently, I have been using it to track the development of the ANSI X3J14 proposed standard. The current state of the standard is reflected in a working document called BASIS. The BASIS changes every three months.

The most recent BASIS is BASIS 10, and I have written ZEN1_10 to match it. ZEN1_10 means Version 1, release 10. I have posted ZEN1_10 on GENIE and on BIX, and will continue to post new versions there.

ZEN1_10 is not meant to be a development system, but rather a simple and efficient Forth dialect. I have provided only the source code, for your study, and an executable file that you can use to load a text file to test a program for ANSI compatibility.

Yes, you are missing documentation, assembler, metacompiler, etc. These will not be written until the draft proposal dpANS is ready, which is at least nine months away. The current release was created by a Forth-to-assembler-source translator. The next release will probably be written in Turbo C or C++.

More on Algorithm Patenting

Dear *DDJ*,

The compression algorithms have been in my conscious path for a search to reduce some of my voluminous writings. I have corresponded with you and Mark Nelson about this, and although I could never get his C program to run with "Let's C" from Mark Williams, I read with interest what some of the law types have to say about it.

Having been in the chemical field for some 30 years, I have come across many snafus of the Patent Office. I leave to your imagination why these snafus occur; not in the least is the heavy burden of research of prior art before patents are granted. Many times patents were granted on chemical procedures or compounds that were in direct conflict with prior art. These were easy to deal with. Usually showing prior art would annul the patent rights right on the spot.

(continued on page 14)

(continued from page 12)

It may have become a bit more difficult today, since our society is the most litigious in the world and lawyers, in and out of government, seem to thrive on perpetuating their own income at the expense of the general population. Lawyers have become the true leeches of this society, that leech wealth from this society. I am not surprised that some two-bit lawyer will claim the LZW routine to be patentable, while the real inventors lived some 50 years ago and may have been dead for a while. After all, lawyers have to make money too.

Paul A. Elias
Fountain Hills, Arizona

Location IS Everything

Dear *DDJ*,

I'm working with Softaid's hardware 8088 emulator, and found Mark Nelson's January 1990 article ("Location is Everything!") on an exe-to-hex locate utility useful and instructive. However, I had to move the STACK segment in his START.ASM file in front of the other data segments to make the locate program behave correctly; this with Borland's TASM 1.0, C 2.0, and TLINK 2.0, which combination I assume uses some slight unanticipated variation of the 5 million sacred ways of ordering segments and groups. Without this change, the stack segment would up, in a test file, a paragraph *after* the rest of the data, and since LOCATE uses this value to figure out where *all* the data is, it wouldn't relocate properly. (This is because — I would figure but heaven only knows — the exe stack record LOCATE uses was actually in fact the genuine offset of the stack, not of some trifling DGROUP, no matter what START.ASM says.)

Once over that minor difficulty I was able, using various C, TASM, and TLINK debugging options, to include line numbers and globals into an output map file which the Softaid SLD (source level debugger) program and utilities could translate, download, and more or less understand — that is, I could step and breakpoint in source (public variables would-up in the wrong place, but I'm sure a little more hacking could fix that). SLD is a great and powerful thing capable of much more, or so I am told, and inasmuch as the Softaid system is thousands of dollars, we're spending a few hundred more for a sophisticated locator program. But it's nice to have an extra emergency tool, and using/fiddling Mr. Nelson's program was just the bit of 8088-in-ROM exercise I needed to get in the mood. Thanks for the help.

J.G. Owen
Fort Salonga, New York

Round and Round We Go . . . Maybe

Dear *DDJ*,

Recently I had the chance to put to use the parametric circle algorithm described in Robert Zigon's article in the January issue of *DDJ* ("Parametric Circles"). Shortly thereafter, I came across Joseph M. Hovanes Jr.'s letter in the March issue, citing the shortcomings of this algorithm when compared to Bresenham's algorithm.

Although Bresenham's algorithm is more efficient, the parametric approach does have several advantages. First, the eight-way symmetry that Mr. Hovanes mentions can be applied when drawing a parametric circle, too. Second, only floating-point additions and multiplications (i.e., no trig functions) are performed inside the loop. If your computer has a floating-point coprocessor, the execution time is within the same order of magnitude as integer arithmetic.

Lastly, if you need to draw only part of a circle (i.e., an arbitrary circular arc), the parametric algorithm can be easily adapted to start and stop where you please. After examining Bresenham's algorithm for quite a while, I'm pretty sure that it can only draw a complete circle, or one of the eight symmetric sectors.

Ben White
Mountain View, California

It's All in the Numbers

Dear *DDJ*,

The major point Michael Swaine makes in his November 1989 "Swaine's Flames" — that we should not blindly accept "numerical" answers is well taken. Unfortunately, in the second example of incorrect use of numeric things, I believe he is in error and John Paulos is correct. In my 15 years hanging around research laboratories, I have always understood two values to be different by "two orders of magnitude" to mean different by a factor of 10^2, not, as he claims, by 10^{100}. If this were the case, the term would not come up very often, since 10^{100} is a *very* large number — about equal to the number of atoms in the universe.

I ran across a better example of incorrect number usage in an IBM ad. This ad states that the footprint of their new printer (291 square inches) is 33 percent smaller than H.P.'s LaserJet (432 square inches). Give or take a square inch, this is correct. However, the ad then concludes from this fact "And that gives you 33 percent more usable workspace." This proclamation, while sounding somehow reasonable, is correct for only one of all possible workspaces.

For example, my computer/printer space is a fairly typical 80 × 32 inches (2560 square inches). If I had a LaserJet, I would have 2560 – 432 = 2128 square inches of "usable" workspace. (Is a printer really useless?) If, according to IBM, I purchase their product to replace the LaserJet, I will have 33 percent more workspace, or 2128 × 1.33 = 2830 square inches more than the area of my table with no printer at all. Good deal, it saves buying a bigger desk!

In fact, if each time I buy an IBM printer, I get 33 percent more workspace, the purchase of 118 of them should give me control of the entire surface of the earth. However, if I need still more room, even if I only purchase one a day, inside of a year I can have the lateral dimensions of my workspace increasing at an average speed greater than light. But that, as we know, would be ridiculous.

Of course, in my example, what really happens is that after the purchase of an IBM laser printer, I would have 2560 – 291 = 2269 square inches of workspace; 2269/2128 = => 7 percent more than before. This is of some benefit, of course, but it doesn't sound very impressive — and the point of using numbers at all is to impress people — right?

Jeffry Stetson
Villigen, Switzerland

I Fought the Law But I Won

Dear *DDJ*,

I was a little bit surprised by Duntemann's One Law of Portability (*DDJ* March 1990): That it's virtually impossible to take source code for an on-line program and recompile it on an entirely different computer with little if any modifications.

Actually, I know that it can be done, since I've done exactly that by switching Ryan McFarland Cobol code between an IBM PC compatible and a minicomputer running Unix. And come to think of it, why can't any higher-level language include verbs which mean "display this on the user's screen" and "place user's keyboard input into this memory location," regardless of whether the code is compiled and executed on a PC, VAX, or 3090?

Jacob Stein
Monsey, New York

DDJ

We welcome your comments (and suggestions). Mail your letters (include disk if your letter is lengthy or contains code) to DDJ, *501 Galveston Dr., Redwood City, CA 94063, or send them electronically to CompuServe 76704,50 or via MCI Mail, c/o* DDJ. *Please include your name, city, and state. We reserve the right to edit letters.*

Generation Scavenging

An efficient, unobtrusive, portable garbage collector

Frank Jackson

Nobody likes to take out the trash, and programmers are no exception. Wouldn't it be nice if someone else would gather up all the garbage that we create and dispose of it for us? It should come as a welcome relief, then, to discover that the run-time systems of programming languages such as Lisp, Smalltalk, and Prolog generally provide facilities that do exactly that. With the advent of powerful computer workstations, automatic garbage collection has become an important component of many modern interactive programming environments as well as the applications that are built using such environments.

Although traditional programming languages such as C, Fortran, and Pascal do not require the programmer to expend any effort managing the memory occupied by either the data that is allocated on the system's run-time stack or the data that is statically allocated on the system's heap, they do require the programmer to manage any data that is dynamically allocated on the heap. Programmers that use such languages are forced to litter their programs with explicit free statements if they wish to recycle the storage consumed by heap-allocated data that is no longer useful. By having the language's run-time system collect such garbage automatically, a certain class of well-known bugs is eliminated. For example, storage leaks cannot occur in such a system, so valuable memory is not wasted if the programmer neglects to free data that is no longer accessible. Even more important, data cannot be prematurely freed, avoiding the chaos that can result when an application tries to access data that was mistakenly recycled. Finally, the programmer is relieved of the burden of having to explicitly manage the heap, which saves development time and results in less complex code.

Frank is a member of the technical staff at ParcPlace Systems and has spent much of his time there designing, building, and evaluating various forms of automatic memory management. He can be reached at ParcPlace Systems, 1550 Plymouth St., Mountain View, CA 94043.

Given these benefits, you might expect heap-based garbage collection to be an integral component of most present day language implementations. This is not the case, however. Most traditional programming languages were not designed with garbage collection in mind, and it is generally difficult to retrofit existing language implementations with an automatic garbage collector. Further, there are a number of serious drawbacks to the classical garbage collection algorithms. These drawbacks include:

- Distracting pauses when performing a garbage collection
- Failure to reclaim certain types of garbage
- High overhead in space and time

These drawbacks become even more apparent when the algorithms are deployed in modern interactive environments, given the stringent response-time requirements of these environments.

Significant progress has been made in the past decade, however, and new garbage collection techniques have been developed that all but eliminate the above drawbacks. One of these techniques — generation scavenging — not only addresses each of the problems just listed, but it requires no hardware support, making it portable across a wide variety of personal computers and engineering workstations. In this article, I'll discuss some of the historical events that led to the development of the original generation-scavenging algorithm. In addition, I'll describe some of the more recent refinements that significantly enhance the performance of the basic generation scavenger. In particular, the scavenging algorithm described later can be tuned so that the average pause time and the total overhead for collecting garbage can be reduced to an acceptably low level.

Classical Garbage Collection Algorithms

Automatic garbage collection has roughly a 30-year history, starting with the near simultaneous invention of the two classical approaches to garbage collection from which most modern collection schemes are derived, namely the mark-and-

sweep collection algorithm and the reference-counting algorithm. In 1960, Collins[2] introduced the notion of using reference counts to determine if a piece of data could be safely reclaimed. Each piece of data, which I shall refer to hereafter as an object, has associated with it a count of the number of other objects that reference it. When this count drops to zero, the object can be reclaimed automatically by the run-time system.

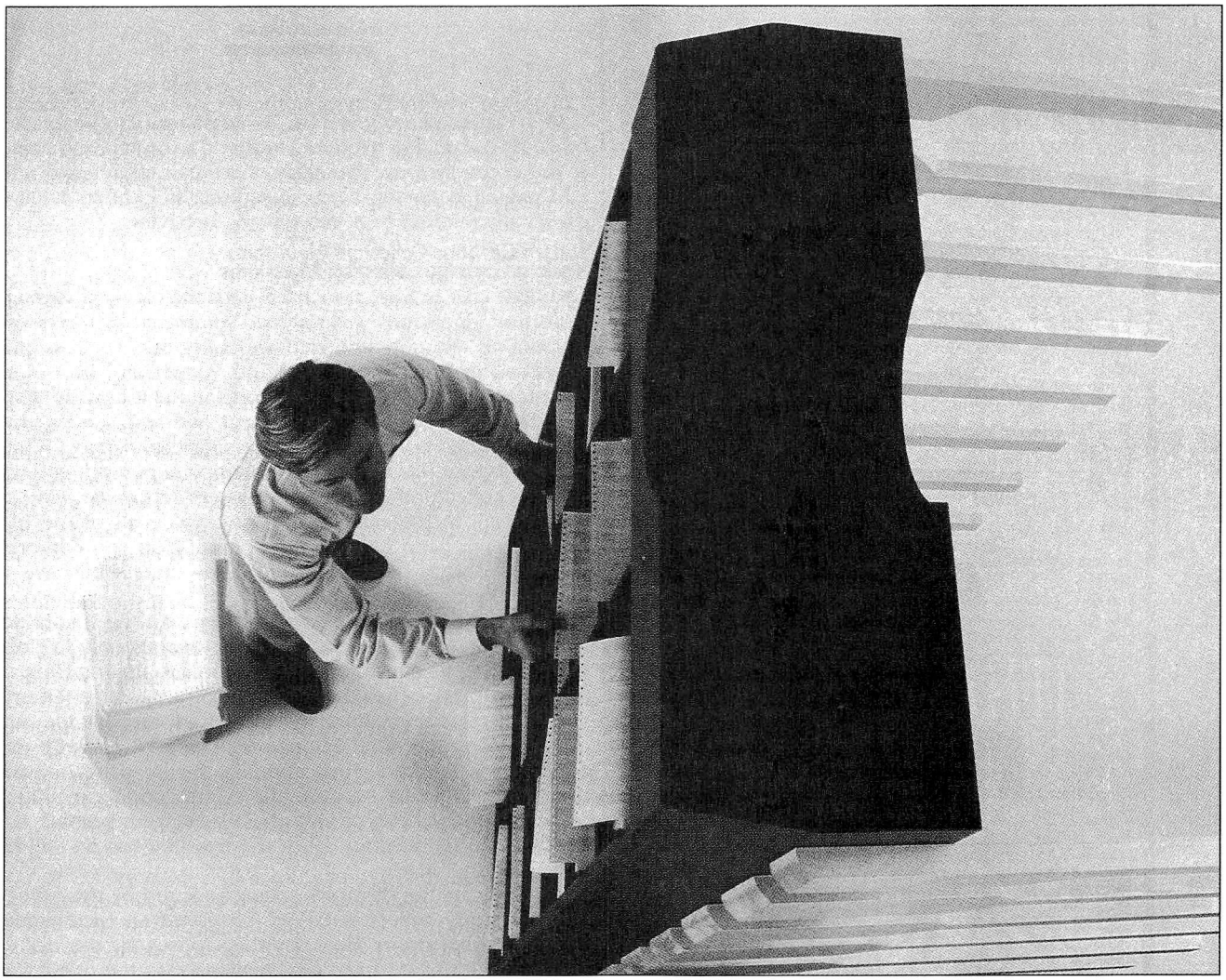
The primary advantage of the reference-counting approach is that the pauses required for such reclamations are generally imperceptible to the user, because these object reclamations can easily be distributed across the computation. In addition, the space occupied by the garbage objects can be recycled immediately, thereby reducing the total amount of memory required to complete a given computation, although this space advantage is reduced somewhat by the storage required to hold the per-object reference counts.

There are some significant disadvantages to the reference-counting approach, however. Most importantly, it can't reclaim circular garbage, because any object that is indirectly self-referential will never have a reference count of zero, even if the object is no longer accessible to those objects that are still involved in the computation. In addition, special provisions have to be made to reclaim those objects whose reference-count fields have overflowed. Accordingly, most systems that employ reference counting attempt to prevent these storage leaks by either providing a backup garbage collection system that utilizes a different collection algorithm or by incurring the expence of additional recursive scanning.

Reference counting also has high overhead because each store requires the run-time system to decrement the reference count of the object whose reference is being overwritten and to increment the reference count of the object whose reference is being stored. In addition, when an object's reference count drops to zero, the run-time system has to decrement the reference count of every object pointed to by the dying object, possibly causing the reference counts of these objects to drop to zero, forcing the system to decrement the reference counts of still more objects. Finally, additional overhead is engendered by the necessity of recycling the storage occupied by these dead objects in order to avoid running short of memory.

Subsequent refinements to the basic reference-counting algorithm by Deutsch and Bobrow[4] in 1976 have succeeded in reducing the total temporal overhead to approximately ten percent, which is still a relatively high price to pay. Consequently, reference counting is no longer widely used in commercially available language implementations. The fact that reference counting permits object reclamation based strictly on local information, however, has made it relevant to systems that must operate in a distributed computing environment.

The other classical garbage collection technique, also proposed in 1960, is McCarthy's[7] mark-and-sweep algorithm. Unlike reference counting, which uses local informa-



Dr. Dobb's JOURNAL
SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

PUBLISHER *Peter Hutchinson*

EDITORIAL
EDITOR-IN-CHIEF *Jonathan Erickson*
MANAGING EDITOR *Monica E. Berg*
TECHNICAL EDITORS *Michael Floyd, Ray Valdes*
ASSOCIATE MANAGING EDITOR *Janna Custer*
CONTRIBUTING EDITORS *Al Stevens, Jeff Duntemann, Martin Tracy, David Betz, Tom Genereaux, Andrew Schulman*
COPY EDITORS *Rhoda Simmons, Pamela Dillehay, Nan Fornal*
EDITOR-AT-LARGE *Michael Swaine*

ART/PRODUCTION
ART/PRODUCTION DIRECTOR *Larry L. Clay*
ART DIRECTOR *Michael Hollister*
PRODUCTION SUPERVISOR *Amy Shulman Lesovoy*
TYPOGRAPHERS *Teresa Raines, Margaret Anderson, Charlene Carpentier*
COVER PHOTOGRAPHER *Michael Carr*

CIRCULATION
DIRECTOR OF CIRCULATION *Maureen Kaminski*
CIRCULATION MANAGER *Randy Robertson*
CIRCULATION PLANNING MANAGER *Manny Sawit*
DIRECT MARKETING MANAGER *Andrea Weingart*
NEWSSTAND MANAGER *Sarah Forsman*
DIRECT MARKETING COORDINATOR *Francesca Davies*
PROMOTION COORDINATOR *Pam Moore*
FULFILLMENT COORDINATOR *Anne Jean*

ADMINISTRATION
VICE PRESIDENT OF FINANCE *Kate Deschamps*
CONTROLLER *Mary Collopy*
CREDIT MANAGER *Betty Arsene*
ACCOUNTING SUPERVISOR *Renate Kernke*
ACCOUNTS RECEIVABLE *Wendy Ho*
ACCOUNTS PAYABLE *LuAnn Rocklewitz*

MARKETING/ADVERTISING
DIRECTOR OF SALES AND MARKETING *Karla Spormann*
ADVERTISING COORDINATOR *Laura Stack Pullen*
MARKETING ASSISTANT *Sara Noah Ruddy*
ACCOUNT MANAGERS *see page 152*

M&T PUBLISHING INC.
CHAIRMAN OF THE BOARD *Otmar Weber*
DIRECTOR *C. F. von Quadt*
PRESIDENT *Laird Foshay*
VICE PRESIDENT OF PUBLISHING *William P. Howard*
VICE PRESIDENT/GROUP PUBLISHER *Randall L. Stickrod*

DR. DOBB'S JOURNAL (USPS 307690) is published monthly, except semimonthly in December, by M&T Publishing, Inc., 501 Galveston Dr., Redwood City, CA 94063; 415-366-3600. Second-class postage paid at Redwood City and at additional entry points.

ARTICLE SUBMISSIONS: Send manuscripts and disk (with article and listings) to the editorial assistant 415-366-3600.

***DDJ* ON COMPUSERVE:** Type GO DDJ.

***DDJ* LISTING SERVICE:** 603-882-1599. Supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Type *listings* (use lowercase) at the login prompt.

SUBSCRIPTION: $29.97 for 1 year; $56.97 for 2 years. Foreign orders must be prepaid, including the additional postage (air or surface) in U.S. funds drawn on a U.S. bank. Add $13 per year for surface mail; add $36 for airmail to Canada and Mexico; or $26 for airlift to all other countries.

POSTMASTER: Send address changes to *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. **ISSN 1044-789X**

CUSTOMER SERVICE: For subscription questions, call toll-free 800-456-1215. For subscription orders or change of address call toll-free 800-456-1215 (U.S. and Canada) or write *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. For book/software orders call 800-533-4372 (in California 800-356-2002).

FOREIGN NEWSSTAND DISTRIBUTOR: Worldwide Media Service Inc., 115 E. 23rd St., New York, New York 10010; 212-420-0588 FAX 212-420-1265.

Entire contents copyright ©1990 by M&T Publishing, Inc., unless otherwise noted on specific articles. All rights reserved.

The Audit Bureau

tion to make its decisions, the mark-and-sweep algorithm relies on a global traversal of all live objects to decide which objects can be reclaimed. The basic mark-and-sweep algorithm works as follows:

1. Mark all objects reachable from the system's roots as being live objects.
2. Sweep memory, unmarking live objects and reclaiming dead objects, possibly performing a simultaneous or subsequent memory compaction.

Although the mark-and-sweep algorithm does reclaim circular garbage, it, too, has serious drawbacks:

- High overhead, because the system has to reference both live and dead objects during each garbage collection.
- Lengthy, potentially disruptive pauses that are proportional to the amount of memory that is currently allocated.

Generation scavenging has proven to be an efficient, unobtrusive technique for reclaiming storage among an object population where deaths outnumber survivors

In a virtual memory system, these pauses can be exacerbated by the paging overhead required to touch each object in the entire system. The mark phase can cause especially bad paging behavior, because it typically exhibits what is essentially random page-referencing behavior.

Modern Garbage Collection Algorithms

The high cost in time associated with the classical garbage collection algorithms was reduced somewhat by the development of the copying garbage collectors, such as that described in 1969 by Fenichel and Yochelson.[5] In its simplest form, a copying collector works in the following manner: The data heap is divided into two semispaces, and object allocations are restricted to a single semispace. When that semispace fills up, the computation is paused, and the garbage collector then traces the system's roots, copying all live objects to the other semispace. Once all of the live objects have been copied, the computation can continue with new objects being allocated in the same semispace as the live objects.

Such a collector is potentially faster than the traditional mark-and-sweep collector because it touches only live objects, making its pause time proportional to the total size of the live objects rather than the size of allocated memory. Dead objects are reclaimed by virtue of the fact that they are not copied to the other semispace. The problem with this approach is twofold: First, dividing the heap into semispaces wastes space because the computation can utilize only half of the heap at any given time, and, second, the pauses required to copy all of the live objects can still be quite lengthy.

To eliminate the disruptive pauses caused by this sort of stop-and-copy collector, Baker[1] proposed an incremental copying collector in 1978. In this approach, the act of copying live objects from one semispace to the other was

(continued from page 18)
interleaved with the actual computation. This algorithm imposed some additional forms of overhead on the computation, however, including the need for the computation to monitor every read and write to the heap in order to correctly follow the forwarding pointers that were placed at an object's old address when it was copied to the other semispace. This additional overhead was typically overcome by using hardware support, such as that available on the MIT Lisp machines. Even so, the Baker collector was just as spatially inefficient as the other copying collectors, because it also divided the heap into a pair of semispaces. In addition, the total time required to copy all of the live objects with each collection was still unreasonably long.

Generation-Based Garbage Collectors

To alleviate the problems associated with these early copying collectors, language implementors began to exploit the empirical properties of data. Researchers observed that young objects tended to die while still young, whereas older objects were more likely to live on indefinitely. It made sense, then, to devote more effort to collecting young objects, where the return on the system's copying investment was likely to be high, instead of repeatedly copying older objects that simply refused to die.

To make such an approach possible, Lieberman and Hewitt[6] proposed in 1983 that objects be segregated into multiple generations, with each generation containing objects of roughly the same age. In addition, they proposed that the system be designed so that each generation could be colleted independently. Younger generations could then be collected more often, generally using some sort of copying algorithm. If an object survived enough collections, then it would be promoted to an older generation, where it would be collected with less frequency. This approach saved time because there were fewer objects to copy in the young generations, given the high mortality rate of young objects; it also saved space because in principle the system needed to maintain only enough free space at any given time to copy the live objects in a single generation rather than enough free space to copy all of the objects in the entire system.

The feasibility of the generation-based approach depended in part on the speed with which the garbage collector could identify the live objects in a given generation. Consequently, the implementors of the generation-based systems developed various methods for keeping track of the roots for each generation. This task was typically simplified by keeping track only of the objects in older generations that pointed directly to objects in the younger generations, and being careful to collect all of the younger generations when collecting an older generation. Nevertheless, the task of keeping track of each generation's roots added additional overhead to the computation. For example, each store into the heap had to be monitored to see if it created the sort of intergenerational reference that increased the number of roots for a given generation. However, the overhead required to monitor each store in such a manner was generally less than the store overhead required by the reference-counting approach. At any rate, the generation-based approach allows the language implementor to stake out an intermediate position between those taken by the reference-counting approach, which relies solely on local knowledge to reclaim garbage, and the mark-and-sweep approach, which has to traverse the entire system to reclaim garbage.

Even more overhead is incurred by those generation-based systems that reclaim garbage incrementally (as noted in the discussion of the Baker[1] algorithm). Such systems,

(continued from page 20)
most notably Moon's[8] ephemeral garbage collector for Lisp, generally hide much of this overhead by taking advantage of the hardware support provided by modern Lisp machines. Such systems are still in use today, and the language implementors on these machines continue to find ways to further utilize these hardware capabilities (for example, Courts[3] has described a way to reduce page faults by dynamically improving the locality of reference of those objects housed on the heap).

Generation Scavenging

Many language implementors, however, especially those developing third-party software that must be deployed on a variety of hardware platforms and operating systems, can't count on having any hardware support for garbage collection at their disposal. Without hardware support, it is quite difficult to implement an efficient incremental garbage collector, primarily because of the extra overhead required to follow forwarding pointers when reading and writing to the heap. The generation-scavenging algorithm was developed to provide language implementors with a garbage collector that, like the incremental collectors, was unobtrusive but, unlike the incremental collectors, did not require hardware support in order to be reasonably efficient.

The basic algorithm, as described by Ungar[10] in 1984, requires that the heap be divided into two spaces — *oldspace* and *newspace*. *oldspace* is generally much larger than *newspace*, because *oldspace* is used as the repository for objects that are considered permanent. As such, *oldspace* is collected infrequently, typically by using a global mark-and-sweep collector. Instead, most reclamation attempts are focused on the objects in *newspace*. In Ungar's[10] original algorithm, *newspace* is divided into three zones — a creation zone and two survivor zones. New objects are allocated in the creation zone. Whenever the creation zone fills up, the computation is halted, and the scavenging mechanism copies all live objects in the creation zone to one of the survivor zones. A forwarding pointer is left behind for each object that is copied, and any other objects that reference the old location of a copied object will have these references updated when the scavenger scans them in its search for additional survivors. Once the scavenge is complete, the creation zone will be empty and can be reused. Subsequent scavenges will continue to copy those live objects that can be found in the creation space and the occupied survivor zone to the empty survivor zone. In this scheme, then, objects are born in the creation zone and thereafter bounce back and forth between the two survivor zones until they are deemed old enough to be promoted to *oldspace*. Ungar[11] refers to this process of promoting an object to *oldspace* as "tenuring."

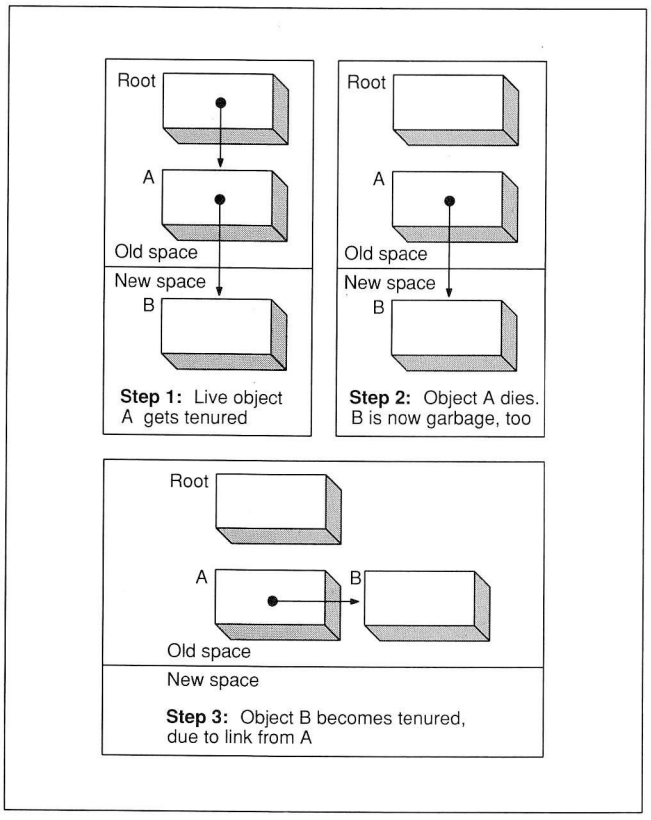


Figure 1: *Nepotism*

Because generation scavenging is a stop-and-copy algorithm, as opposed to being incremental, there is no need for special hardware to follow forwarding pointers, because all references to these forwarding pointers are automatically updated during the scavenge. Like the other generation-based algorithms, however, the system must maintain a list of roots for *newspace*. It must keep a list of those objects in *oldspace* that contain references to objects in *newspace*. Maintaining this list imposes some extra overhead on every store into the heap. Even without special purpose hardware, however, this overhead doesn't appear to be onerous for the following empirical reasons:

- Stores occur much less frequently than fetches.
- Very few *oldspace* objects point to objects in *newspace*, so the list of roots is generally small.
- The vast majority of stores (that is, the stores into those objects housed in *newspace*) can be ignored on the basis of a single boundary check.

In addition, the system must be able to discern the age of an object so that the scavenger can decide when the object should be tenured to *oldspace*. In Ungar's[10] initial implementation, each object had an age field that the scavenger incremented periodically. As we shall see shortly, the requirement that the system keeps track of each object's age need not impose any additional storage overhead. Generation scavenging, then, can be viewed as a two-generation system, where *oldspace* and *newspace* are the two generations, or a multi-generation system with objects of different generations (that is, different ages) being housed together in *newspace*.

Because the generation-scavenging algorithm was first published, many variations have been both proposed and implemented. In some systems, *newspace* is composed of two zones instead of three. Other implementations allow the scavenger to scavenge multiple spaces instead of restricting its purview to a single space. These spaces are sometimes arranged as pairs of semispaces and sometimes as a bucket brigade of consecutive spaces through which the surviving objects are promoted. Some systems eliminate the need for an age field by spatially segregating objects of the same age. (See Wilson and Moher[13] for an example of a system that obviates the need for an age field by organizing its spaces into a bucket brigade.) Finally, various schemes have been proposed for efficiently identifying the roots of *newspace*. Shaw,[9] for example, recently suggested combining the store check with the virtual memory mechanism that marks hardware pages as being dirty and then scanning these dirty pages for actual roots at scavenge time.

Tenure Policies

One of the key decisions that a generation scavenger must

make is when to tenure an object to *oldspace*. Early scavengers generally employed a simple fixed-age tenure threshold: They tenured any object that had survived for a fixed amount of time or a fixed number of scavenges. Studies conducted by myself and Ungar[12] show that such tenure policies are not particularly effective in minimizing the amount of tenured garbage (that is, objects that die after being tenured) or in controlling the length of the pauses required to perform the scavenge. Different applications cause objects to survive for different amounts of time, so no single tenure threshold will perform optimally in all circumstances. If the tenure threshold is set too young, then *oldspace* will be flooded with objects that die shortly thereafter. (This problem will be further exacerbated by the effects of nepotism. See Figure 1.) And if the tenure threshold is set too high, then the scavenge pauses can easily become disruptive.

By having the language's run-time system collect garbage automatically, a certain class of well-known bugs is eliminated

Both of the above problems can be solved by employing a tenure policy that modifies its tenure threshold dynamically according to the demographics of the object population currently housed in *newspace*. I will now describe how one might go about designing such a tenure policy. Because stop-and-copy collectors traditionally have problems with distracting pauses, it is important to provide the scavenger with the means to control the length of its pauses. Assuming that we have determined the maximum pause time that the scavenger can be permitted to take without being considered disruptive, we need to measure how many bytes of surviving objects the scavenger can copy in that amount of time. The scavenger can then control the length of its pauses by using this number as a watermark in the survivor zones. If the aggregate size of the objects in the survivor zone is less than this watermark, then the scavenger doesn't need to tenure any objects during the upcoming scavenge, because the pause required to scavenge these survivors will probably be acceptably brief. If, however, the size of these survivors actually exceeds this watermark, then the scavenger should tenure some objects during the next scavenge to keep the pause times from becoming disruptive.

Because the scavenger tenures objects to keep the duration of its pauses under control, we need to provide it with the means to minimize the amount of the tenured garbage that it creates. Rather than tenure objects randomly, which could result in young objects that are unworthy of promotion being tenured, the scavenger uses demographic information to select a tenure threshold that will result in the desired amount of the oldest objects in *newspace* being tenured. The necessary demographic information can be kept in a table indexed by age that contains the number of data bytes in *newspace* for each age. This table can either

(continued on page 26)

(continued from page 23)

be maintained by the scavenger as a matter of course, or it can be created on-the-fly by a quick scan of the occupied survivor zone. By scanning this table backwards, the scavenger can then set the appropriate tenure threshold for the ensuing scavenge. These measures permit the scavenger to tenure the minimal amount of objects that have the highest likelihood of surviving (see Figure 2).

Thus far, we've described a scavenger that can easily be made nondisruptive, even in the face of the varying object demographics, but what about the total scavenging overhead? Given a maximal acceptable pause time of, say, 100 milliseconds, we can drive the total scavenge overhead reasonably low by sizing the creation zone appropriately (assuming that the overhead required to perform the store checks is as low as recent studies seem to suggest). That is, if we were to size the creation zone such that it filled up once per second (and, hence, a scavenge was performed every second or so), then the total overhead for scavenging would be around ten percent. If, however, we sized the creation zone so that it filled up every three to four seconds, then the scavenge overhead would be less than three percent.

Thus, the generation scavenger described can easily be tuned in two respects: The average pause time required to perform a scavenge and the total scavenge overhead can be controlled by setting the watermark in survivor space and the size of the creation zone, respectively. Of course, the cost for reducing both pause times and scavenge overhead is paid in memory, in terms of both the memory required to size the creation zone appropriately and the space taken up by tenured garbage resulting from the need to keep the total size of the scavenge survivors less than the survivor zone watermark. For example, current Smalltalk implementations on stock hardware that utilize this particular type of scavenger typically have survivor zone watermarks that vary between 50K and 120K, resulting in worst case pause times of 100 milliseconds, and creation zones between 400K and 800K,

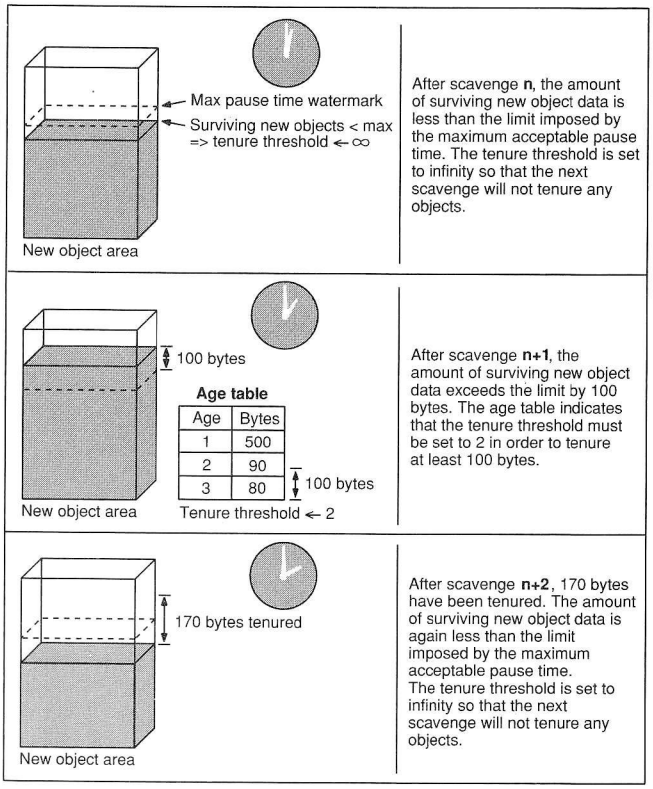


Figure 2: *Demographic feedback-mediated tenuring*

(continued from page 26)
resulting in a scavenge overhead of less than three percent.

Because it requires neither hardware assistance nor any operating system's software, this scavenger has been successfully deployed as a component of Objectworks for Smalltalk-80 fielded by ParcPlace Systems. This particular implementation, coded entirely in C, has been ported to a wide variety of personal workstations, including the Apple Macintosh family, most 386-based DOS PCs, and the workstations sold by Sun, Digital Equipment, and Hewlett-Packard.

Trouble in Paradise

Generation scavenging is not without its shortcomings, however. Certain space-consumptive programs may produce so many live objects that their sheer volume simply overwhelms the capacity of the *newspace* architecture described earlier. These programs typically produce substantial amounts of tenured garbage that can result in wasted memory, poor paging behavior, and lengthy interruptions required to reclaim this garbage. These problems can be mitigated somewhat by utilizing several large additional generations, but not without incurring noticeable pauses when these additional generations are scavenged with the current generation of stop-and-copy algorithms. Finding an efficient way to collect these additional generations without perceptible pauses on stock hardware will require further research.

Furthermore, real-time programs frequently require drastically shorter scavenge pauses than normal interactive programs. The measures required to reduce these pauses can also result in a significant increase in the amount of tenured garbage.

Conclusions

Generation scavenging has proven to be an efficient, unobtrusive technique for reclaiming storage among an object population where deaths outnumber survivors. In addition, it has proven to be popular among language implementors. For example, all of the commercially available Smalltalk implementations use variants of generation scavenging as their primary reclamation systems. This popularity can be attributed both to the algorithm's simplicity and to the fact that it requires no special hardware support. Finally, I expect future developments in the area of generation-based garbage collection to proceed along the following lines:

- Addressing the problems posed by programs that produce massive quantities of intermediate-lived objects.
- Finding ways to reduce the number of page faults in programs that utilize virtual memory.
- Searching for more efficient methods of keeping track of the roots to a given generation.
- Experimentation with alternate tenure policies.

References

1. Baker, Henry G., Jr. "List Processing in Real Time on a Serial Computer." *Communications of the ACM* 21(4) (April 1978).
2. Collins, George E. "A Method For Overlapping and Erasure of Lists." *Communications of the ACM* 2(12) (December 1960).
3. Courts, Robert. "Improving Locality of Reference in a Garbage-Collecting Memory Management System." *Communications of the ACM* 31(9) (September 1988).
4. Deutsch, Peter L. and Bobrow, Daniel G. "An Efficient, Incremental, Automatic Garbage Collector." *Communications of the ACM* 19(9) (September 1976).
5. Fenichel, Robert R. and Yochelson, Jerome C. "A Lisp Garbage Collector for Virtual Memory Computer Systems." *Communications of the ACM* 12(11) (November 1969).
6. Lieberman, Henry and Hewitt, Carl. "A Real-Time Garbage Collector Based on the Lifetimes of Objects." *Communications of the ACM* 26(6) (June 1983).
7. McCarthy, John. "Recursive Functions of Symbolic Expressions and Their Computations by Machine," part I. *Communications of the ACM* 3(4) (April 1960).
8. Moon, David A. "Garbage Collection in a Large Lisp System." Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, pages 235-246, Austin, Texas, August 1984.
9. Shaw, Robert A. *Improving Garbage Collector Performance in Virtual Memory.* Technical Report CSL-TR-87-323. Stanford: Stanford University, March 1987.
10. Ungar, David. "Generation Scavenging: A Non-disruptive High-Performance Storage Reclamation Algorithm." Proceedings of the ACM Symposium on Practical Software Development Environments, Pittsburgh, Penn., April 1984.
11. Ungar David. *The Design and Evaluation of a High-Performance Smalltalk System,* AMC 1986 Distinguished Disertation, MIT Press, Cambridge, Mass., 1987.
12. Ungar, David and Jackson, Frank. "Tenuring Policies for Generation-based Storage Reclamation." OOPSLA'88 Conference Proceedings. ACM, September 1988.
13. Wilson, Paul R. and Moher, Thomas G. "Design of the Opportunistic Garbage Collector." OOPSLA'89 Conference Proceedings, ACM, October 1989.

Figures 1 and 2 originally appeared in the OOPSLA'88 Conference proceedings, ACM, September 1988.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 1.**

Dynamic Link Libraries For DOS

Running large programs in small memory space

Gary Syck

With the average size of available memory increasing, the average size of applications that use that memory also grows, as programmers come up with new ways to put all that memory to use. The real challenge is to shoehorn the great ideas for tomorrow's programs into the limited memory of today's computers. At some point you just can't shrink programs without leaving out attractive features. Of course, you can try to convince users that they don't need their favorite features, but a better strategy is to come up with some kind of memory management that lets you move portions of the program and data in and out of memory as required.

Most PC programming systems provide some way to do overlays, which are portions of a program that load into the same area of memory. When a routine is required, that routine gets loaded into the overlay area, replacing whatever code or data is already there.

Things get complicated when the routine in the overlay area requires a routine that is in another overlay. In this case, the calling routine will be swapped out. This causes problems when the called routine returns and finds something else where the calling routine used to be.

Gary is an independent consultant and can be reached at 12032 100th Ave. N.E., #D203, Kirkland, WA 98034.



To prevent this problem when laying out the overlay areas for a program, you must consider what routines are called from where. Figure 1 shows a typical memory map for a program that uses overlays.

Enter Dynamic Link Libraries

Dynamic link libraries (DLLs), which were introduced with Windows and OS/2, move the task of linking in code from the end of the compile/link step to when the program is actually running. Run-time linking lets the linker put an object file anywhere in available memory. If there is no memory available, the linker can find an object file that is not being used and replace it with the file it needs to load. But to take advantage of DLLs, you must use an operating system that supports them, such as Windows or OS/2. (It is ironic that to make use of a system that makes large programs run in small memory spaces, you must use an operating system that requires several megabytes of RAM to work.)

The routines presented in this article show how to make a run-time linker/loader that uses about 5K of code. This is the heart of a DLL system. At the end of the article are some comments on how this can be expanded into a full DLL system.

(continued from page 30)

The key to understanding what is going on is to remember that a DLL loader is simply a linker that works at run time, combined with a program loader. A conventional linker combines all of the object files for a program into a single file and fixes all of the memory references to point to locations in the file. The loader puts the file in memory, adds the address where the file was loaded to the memory addresses, sets up the segment registers, and jumps to the start address of the program.

I made some assumptions in order to simplify the process of linking and loading object files. The basic assumption is that the object file was created by Microsoft C 5.1, using the large memory model. This choice of language and memory model means that the linker loader will always deal with the same arrangement of segments.

There are three major pieces to this program. The first piece, GETDATA.C (Listing One, page 104), loads all of the global data and creates the master symbol table. The second, STUB.ASM (Listing Two, page 104), is the calling mechanism for functions. LINKER.C (Listing Three, page 105) is the linker itself. Finally, the file DLL.C (Listing Four, page 108) ties them all together.

Loading Global Data

Global data is a bit of a problem for the run-time linker. All global data must be defined before it can be referenced. With conventional linkers this is not a problem because the linker reads all of the object files before doing any fix-ups. The run-time linker looks at one module at a time, so modules that define data must be loaded before modules that use the data.

Another problem with global data occurs when the linker wants to remove a module from memory. Ideally, it should also remove any static data for that module. There is, unfortunately, no way to tell the difference between static data and initialized global data.

The solution to these problems is to require all global data to be defined in a single object file. Loading this object file first insures that global data will be ready when it is needed. Any global data that is not in this file is assumed to be static data associated with the module being loaded. The file used in this sample program is named GBLDATA.OBJ, which is a standard object file that is created by compiling the C source file, GBLDATA.C (Listing Five, page 108), which contains global data definitions. Listing Six (page 109) is the file MAIN.C from which MAIN.OBJ is created; Listing Seven (page 109) is DLL.H, the header file.

The function *GetData* in GETDATA.C reads GBLDATA.OBJ and extracts the data information in it. The function begins by opening the object file. Next, a loop reads each object record from the object file. In the loop is a switch statement that does different things to different record types. When the loop is finished, the linker closes the file and allocates space for the last group of uninitialized data variables.

Two types of variables can be expected in GBLDATA.OBJ. The first type is initialized data. In this implementation all initialized data goes in the near data segment in the *DataSpace* array defined in STUB.ASM.

The PUBDEF record in the object file contains all of the names of the initialized data variables. This information, along with the size and location of the data, is copied into the Syms table. The data that initializes the variables is in the LEDATA records.

The second type of variable is the

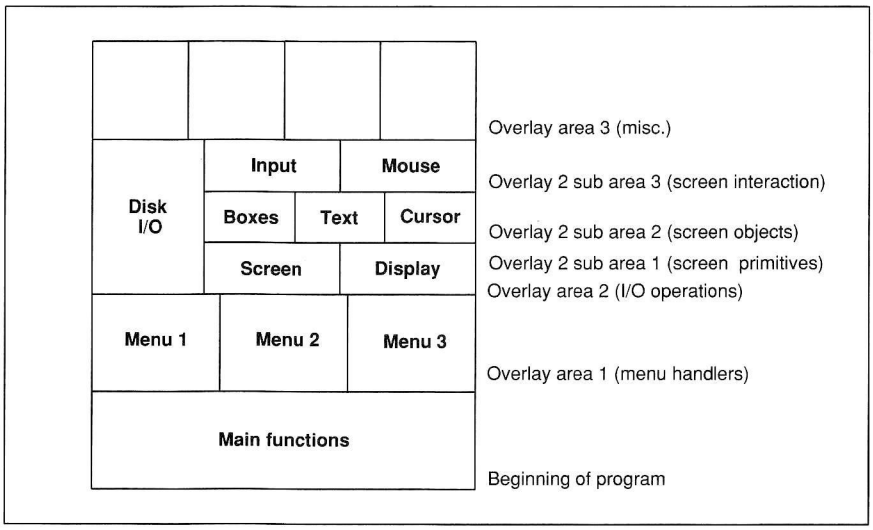


Figure 1: *A typical memory map for a program that uses overlays*

(continued from page 32)
uninitialized data. The names of uninitialized variables can be found in COMDEF records. The COMDEF record contains the name, number of elements, and size of an element for a variable. This information is copied into the Syms table. To reduce the number of allocations required, no allocation

The routines presented in this article show how to make a run-time linker/loader that uses about 5K of code

is made until there is at least 32K of data to allocate. The function *AllocateSyms* does the allocation and puts the appropriate addresses into the Syms table entries for the symbols affected. A final allocation must be done when there are no more records to pick up the last group of variables.

Calling Functions

When *GetData* has loaded all the global data, the main routine moves on to functions. Function information is kept in a table called "FuncLst." Library functions must be placed in the table explicitly (unless you have the object files for the library available). The library functions will be linked to the main program by the conventional linker. In this example the library function *printf* will be used, so it must be placed in the table.

All other functions will be loaded by the run-time linker. Before it can be loaded, a function must have an entry in the FuncLst table. The entry contains the name of the function, the location of the function (or NULL if the function is not loaded, yet), a flag that tells if the function is unloadable, and a copy of the stub routine. The entry after all of the library routines is for the first function in the program. Note that the address of the function is NULL. This indicates that the function has not been loaded yet. To start the program the stub routine for this function is called.

The stub routine is a copy of the function *Stub* in STUB.ASM. Each copy is modified so that it knows what function it goes with. The stub routine will be called instead of the related function. The job of the stub routine is to

(continued from page 34)
determine if the function is in memory (by the value of the address in FuncLst) and load it if it is not. In a system that swaps functions in and out of memory, the stub routine would also set flags indicating that the function is in use or not. Once the function has been loaded, the stub routine jumps to the function.

Linking Modules

When the function is not in memory, the stub routine calls the linker. The first step is to find the file. The name of the file is extracted from the name of the function. If there is only one file, the name of the function must match the name of the file. If there are several functions, they each begin with the name of the file, followed by an underscore, then the function name.

The logic for loading the file into memory is similar to that used for loading GBLDATA.OBJ. The difference is that there are a few more records to deal with. The first new record is the EXTDEF record. This record contains the external symbols used by the module. If the external symbol is in Syms, then it is a data reference; otherwise, it is another function, and an entry is made in FuncLst. Each external item is put in a table called "ExtSyms." This table maps the symbol number used in this file to the symbol numbers used in Syms and FuncLst.

Compiler Supported DLLs for DOS

Andrew Schulman

The main executable for Jensen & Partners International's (JPI) new TopSpeed C development environment, TS.EXE, is only 7K bytes in size. TSC.EXE, the command-line version of the C compiler, is also only 7K. The only .OVL (overlay) file is 38K. So where's the code?

Regrettably, JPI has not returned us to the days of Turbo Pascal 3.0, when a blazingly fast compiler and full-screen text editor fit in under 40K. Instead, most of the JPI environment is contained in files such as TSMAIN.DLL, TSLINK.DLL, TSMAKE.DLL, and TSASM.DLL.

Microsoft's EXEHDR utility (included with the Windows and OS/2 software development kits) reveals that these are "true" segmented-executable dynamic-link libraries (DLLs). The entry points have names such as STR$CARDTOSTR and WINDOWS$PUTONTOP, corresponding to functions from JPI's Modula-2 compiler (for example, *Windows. PutOnTop( )*). This C compiler and development system has, with the exception of library functions such as *printf( )*, been written not in C, but in Modula-2.

JPI's use of DLLs for DOS makes sense for a number of reasons:

- Because the TopSpeed development environment is comprised of individual programs, and because these programs have many subroutines in common, why copy the subroutine from .LIB into each executable? Instead, all programs share a single copy of a subroutine, which stays in a DLL. The programs are clients of the DLL. Database programmers have always known that it is evil to maintain multiple copies of the same piece of data. So why is it okay to have multiple copies of the same piece of code?
- DLLs don't just provide code sharing on disk (which is a capability also offered by non-DLL systems such as PocketSoft's .RTLink). Another benefit is reduced memory consumption: In JPI's implementation of DOS DLLs, as a DLL is loaded into memory, least-recently-used DLLs may be unloaded. All of this takes place without the knowledge of the programmer. Thus, JPI's DOS DLLs act as transparent overlays.
- For JPI, a key reason to use DLLs is that they facilitate plugging new programming languages into TS, which is a language-independent development environment. To add a new language, simply install some DLLs and some header files. In addition to C, 8088 assembler, and Modula-2, JPI plans to add C++ and Ada. Only time will tell what success JPI has with this ambitious plan.
- One consideration is portability between OS/2 and MS-DOS. The OS/2 versions of JPI's compilers use the DLL mechanism provided by OS/2. The much smaller MS-DOS operating system does not provide DLLs, but clever programmers have found that, what MS-DOS does not provide, it also does not prevent. Because the DOS version of the TopSpeed compiler runs in real mode, JPI could not use the Intel "segment not present" exception (INT 0B), which aids dynamic linking in protected mode. While OS/2 supports DLLs, DOS merely allows them.

Purchasers of the TopSpeed C Extended Edition (which costs $200 more than the Standard Edition) get to use the same DOS DLL technology in their own programs. The Extended Edition includes complete source code for JPI's implementation of DOS DLLs.

In contrast to the Microsoft tools, no .DEF file is required to make a DLL in the JPI environment. Instead, you place a directive such as "make DOS DLL," "make OS2 DLL," or "make WIN DLL" (for Microsoft Windows) in a project file. To use DLLs in an MS-DOS program, you use the directive "make DynaEXE."

Analogous to the LIBPATH statement in an OS/2 CONFIG.SYS file, when using JPI DLLs under MS-DOS, you need to set a LIBPATH DOS environment variable.

JPI appears to have given far greater thought than Microsoft to issues of C language support for dynamic linking and multi-threaded programming. For example, the JPI license agreement contains a well thought-out statement regarding distribution of DLLs. Similarly, each function in JPI's Library Reference manual contains a discussion of "Multi-thread considerations." This is connected with dynamic linking, because all DLLs, and all programs that call DLLs, must use JPI's "MThread" model.

It was said earlier that JPI's DOS DLLs are like transparent overlays. When a feature is said to be transparent, it generally means that (to the programmer) it looks like the feature isn't there. However, transparency isn't always useful. In the case of dynamic linking, sometimes the programmer needs explicit control over this process (see my article "Linking While

Andrew Schulman is a contributing editor to Dr. Dobb's Journal *and is a co-author of the book,* Extending DOS *(Addison-Wesley, 1990).*

The other new record is the FIXUP. These records are used to put the addresses of functions and data into the code in the module. Each fix-up refers to an item in Syms or FuncLst (via ExtSyms). The fix-up code looks up the address required and places it in the code at the location specified.

Two of the record types used in GBLDATA.OBJ have a slightly different purpose in the linker. The first is the PUBDEF record. The only public sym-

the Program is Running," *DDJ*, November 1989). Unfortunately, JPI's DOS implementation currently does not provide any way to dynamically link under explicit program control, in that there is no DOS equivalent to the OS/2 functions *DosLoadModule( )* and *DosGetProcAddr( )*. In another sense, though, JPI's implementation of DOS DLLs is currently not transparent enough: You will notice a performance penalty. In one test, a program that called code in a DLL took four times as long to run as the fatter "stand-alone" version that didn't use DLLs. However, JPI claims this was a worst-case example, and that typical DLL use presents only a 10–15 percent performance penalty.

JPI is not alone in bringing DLLs to DOS: Several other companies are working towards the same goal. First and foremost a Modula-2 company, JPI, like other Modula-2 companies such as Stony Brook, is also working to ensure that multi-threaded programming is portable between OS/2 and MS-DOS (multiprocessing is a fundamental part of the Modula-2 programming language). By making DLLs and multi-threaded programming available to MS-DOS C programmers, JPI is helping to bridge the gap between MS-DOS and OS/2.

This is part of a general trend toward making OS/2 features available under MS-DOS. Protected-mode DOS extenders are another example of this trend: They provide a large address space, virtual memory, and protected mode, while still holding onto the venerable MS-DOS operating system. This, like the goal of dynamic linking under MS-DOS, in turn reflects the human instinct to "have your cake and eat it too."

bols in these files should be function names. The PUBLIC record is the only place to get the offsets of the functions in this module. This information will be copied into FuncLst when the code for the function is loaded.

The other record that is different is the LEDATA record. In these files the LEDATA record contains either initialized data or code. The segment field is

DLLs move the task of linking in code from the end of the compile/link step to when the program is actually running

used to tell the difference. Segments are referred to by number in LEDATA records. The number points to a name in the SEGDEF record. Because Microsoft C always uses the same segments, the numbers don't change — so the SEGDEF record does not have to be used. LEDATA records for segment 2 are string literals; segment 3, constants; and segment 1, code.

The string literals and constants are placed in the data segment. The variables DataSize and LocalSize are the start of the local data and the size of the local data. These variables can be used to remove the local data from memory when the function gets swapped out.

The linker allocates space for all of the code segment LEDATA records found. Once the space is allocated, the address of the space and the offset from the PUBDEF record are combined to make the address of the function. This information is then placed in the FuncLst table for use by that function's stub routine.

When there are no more records, the linker closes the object file and updates the DataSize pointer. The linker returns control to the stub routine. The stub routine jumps to the function. It jumps instead of calling the function so that the stack will not contain an extra return address. In a fully functioning DLL system the stub routine would pop the return address off the stack and then call the function. When the function returned, the stub routine could adjust any flags required and return to

the address it popped off the stack.

Summary

DLLs provide an easy-to-use, flexible system for running large programs in small memory spaces. If history is any guide, then there will always be a need for this kind of technique. The implementation of a DLL system is not very complex. You do not need to have a sophisticated operating system to implement one.

The program presented here does the hard part — linking the functions. To make it a complete DLL system you need a memory management system that keeps track of which functions are active and which can be removed from memory.

The program assumes that everything is correct. It will crash if there are undefined symbols or missing object files. This could be corrected with a preprocessor that verifies that all of the pieces are correct. This preprocessor could also put the object files in a single file for easier control.

Using DLLs can change the way you look at programming problems. If the modules are designed well, the same module can be used by many programs. Changing the module changes all of the programs without having to relink every program. Users can customize their software in their own favorite language without having access to proprietary portions of the main program.

Try some experiments with DLLs. You will find your own interesting uses for them, and may well eliminate complaints about how large your programs have grown.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 104.)

Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

Getting a Handle on Virtual Memory

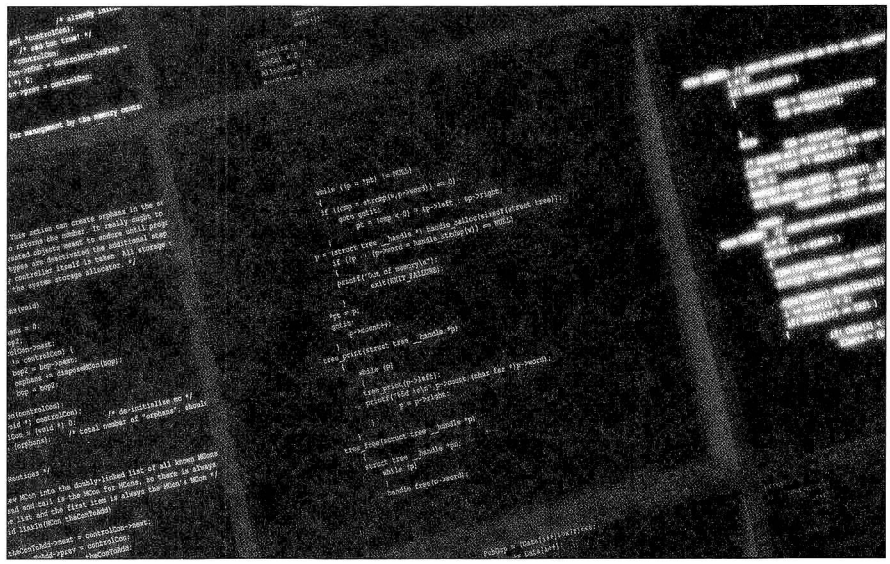
Virtual memory management should be supported by the compiler

Walter Bright

Programmers are well aware that there are serious memory limitations when programming under MS-DOS on the PC. These problems are variously called "ram cram," "the 640K barrier," "brain-damaged 8086" — and occasionally even more colorful terms. There are several different strategies for dealing with this memory limitation problem, including the implementation of virtual memory managers.

Most existing software-based virtual memory managers are clumsy to use: They're prone to obscure, and severe bugs that can make regular pointers in C look simple. Furthermore, the conventions of these packages had to be rigorously adhered to. Even if the program using a virtual memory manager finally got debugged, the results were inefficient and the syntax was aesthetically ugly. Ugly syntax is a sure sign of a poor solution.

A better solution to virtual memory is to have the compiler directly support it. A C or C++ compiler could support a new pointer type called a "handle." The syntax for accessing memory referenced by the handle would then be taken care of by the compiler. This article describes the implementation of handles for expanded memory under MS-DOS using Zortech C/C++.



Walter is the director of the compiler development division at Zortech Ltd. He has a degree from Cal Tech and can be reached at 4819 118th Ave. N.E., Kirkland, WA 98033.

Handle Pointers

The handle refers to dynamically allocated data, serving the same general purpose as a regular pointer. The difference is that while data pointed to by a regular pointer is allocated on the heap, the data pointed to by a handle can reside in expanded memory, extended memory, or a disk sector.

To actually refer to the data, the handle must be converted into a pointer by a function. The function extracts a logical page number and an offset from the handle; the logical page is then swapped into a physical page, and the offset is added to the physical page address. The result is returned as a pointer. This process simulates virtual memory swapping in software.

For handles to be useful, they must adhere to the following criteria:

- Handles must be easy and natural to use.
- The compiler must do as much of the work as possible.
- Handles should be implemented as a special pointer type.
- Porting programs using handles to computers that directly support virtual memory must be easy.

(continued from page 40)

- The source code, when ported to a virtual memory machine, must run as efficiently as if it had been written with conventional pointers.
- There should be no special functions to call to access a handle.
- The initialization and termination must be handled automatically.
- The behavior of handles must be adjustable via library routines.
- Handles should be upwardly compatible with C++. (Because C++ is the wave of the future, adding an extension that will be ugly to work into C++ would be a major impediment.)
- The handle must be a compatible extension to ANSI C.

How will handles look in source form? With PC C compilers, it is already common to support multiple pointer types with the syntax shown in Figure 1.

Supporting handles in the same style, then, suggests the following syntax:

```
long _handle *h;    /* h is a handle
                       to a long */
```

The keyword *_handle* was chosen because it is compatible with name space rules for extensions to ANSI C. The underscore convention is used because, as it turns out, the identifier handle is used by a lot of existing code. Finally, the keyword virtual would conflict with C++ use of that keyword.

Implementation

A handle is a 32-bit type. The high 16 bits refer to the page, in a manner defined by the library implementation. The low 16-bits form the offset into that page. Obviously, this restricts the page size to less than 64K in length. Handles are unique; no two handles can refer to the same location in handle space.

Pointer arithmetic with handles follows the same rules and behaviors as does far pointer arithmetic (that is, only the 16-bit offset is manipulated). Comparisons are also handled the same way as for far pointers, that is, for < <= >= and > the comparison is done for the 16-bit offset only. For = = and !=, comparison involves the full 32-bit value.

The handle refers to dynamically allocated data, serving the same general purpose as a regular pointer

The high 16 bits of a handle refer to the page where the data is stored. Because it's desirable to avoid a performance penalty if expanded memory is absent, the format of the page reference is a bit tricky. If expanded memory is not present, handles will point into real memory. The best way to do that is with a far pointer, the handle format must be able to distinguish a handle from a regular far pointer.

The 8086 has a 20-bit address space, of which the high 16 bits form the segment. All possible 16-bit segment values are valid segment values, so there isn't a simple way to distinguish a handle from a far pointer. On the PC, the ROM BIOS occupies the high end of the address space. Programs almost never access that part of the address space (and if they do, they can use a regular far pointer). Valid handle values are only manipulated by library functions, and are only created for dynamically allocated data.

Segment values from 0×FE00 to 0×FFFF can therefore be defined as handles. (This is controlled and is adjustable by the library portion of the han-

dle support.) This yields 256 pages. Because this is the number of pages that will be allocated in expanded memory, and each expanded memory page has 16K bytes in it, 256 pages make for a total of 4 Mbytes of handle memory space. This ought to be sufficient space for most applications, and you can extend the range of handles to more than 0×FE00 to 0×FFFF.

It's the library implementation's job to distinguish a handle from a far pointer. The segment portion is compared against 0×FE00. If the 16-bit segment is less than 0xFE00, the handle is a regular far pointer. Otherwise the upper 16 bits of the handle refer to logical page number (segment -0×FE00).

Conversions from handles to far pointers occur whenever a handle pointer is dereferenced or when a handle is cast to a far pointer. The conversion process is done by a library routine that swaps the logical page into the address space and returns a far pointer into it. Figure 2 shows examples that do handle conversions to far pointers. Conversions from far pointers to handles are just a type of 'point,' that is, no change in the bit pattern.

The optimizer is aware that handles are a special type. Its main job is to determine when a new handle dereference is necessary and when a previous conversion can be used instead. Consider the example in Figure 3. Clearly, *h* only needs to be converted to a far pointer once. It converts the code to that shown in the second portion of Figure 3.

The result of a conversion in Figure 3 cannot be used if:

1. The value of the handle might have changed.
2. A handle dereference was done on another handle (thus possibly overwriting the previous page with another).
3. A function was called (because that function may convert other handles, resulting in case 2).

Because handle memory is larger than physical memory, pages from handle space are swapped into buffers in physical space. Each handle conversion may result in a new page from handle space overwriting a previous page, invalidating any pointers into that page. This explains the reasoning behind assuming that any handle conversion invalidates any previous conversions.

Of course, you can always convert a handle to a pointer yourself. If you know that a function call does not convert any handles, then the conversion is still valid after the function call. Also, if the library implementation of handle conversions uses more than one physical page (the expanded memory version described shortly uses four), you can rely on at least that many conversions being valid simultaneously.

Handles and Expanded Memory

Let's look at how Zortech C and C++ implement handles. It turns out that expanded memory is particularly well suited for implementing handles. It is fast; swapping a logical page into a physical page usually requires a simple write into an I/O register. Expanded memory can be efficiently emulated on 386 machines by using the memory mapping features of the CPU. (QEMM-386 from QuarterDeck is an example of this type of utility.) There are even

```
void *p;                /* pointer type is default for the memory model */
char far *pc;           /* far (segment and offset pair) pointer        */
int near *p;            /* near (offset only, segment is assumed)       */
```

Figure 1: *Typical pointer types used in most PC implementations of C*

```
int _handle *h;
struct A _handle *h2;
int far *f;
int i;
extern void func(int far *pi);

f = h;
*h = i;
h[3] = *f;
i = *(h + 6);
h2->b = i;
func(h);
h = (int far *) h;
```

Figure 2: *Converting pointers from handle to far*

```
        struct { int a,b; } _handle *h;
        h->a = 1;
        h->b = 2;
/* Converted code */
        struct { int a,b; } _handle *h, far *p;
        p = h;
        p->=a = 1;
        p->=b = 2;
```

Figure 3: *Example showing that the optimizer is handle aware*

emulators that fake expanded memory by using extended memory (memory above 1 Mbyte) or even a hard disk. A final impetus for using expanded memory is that it is part of DOS 4.0.

The expanded memory concept is that one or more logical pages of bank-switched memory are mapped into real memory (physical pages) as needed. This corresponds directly to the handle concept of a large amount of memory (handle space) of which a subset is mapped into physical memory buffers. The mapping of a logical page to a physical page occurs when a handle is dereferenced.

Expanded memory makes it possible to have two physical pages actually map onto the same logical page. In other words, expanded memory allows two different addresses to refer to the same memory location! This implementation of handles carefully avoids depending on this "feature," as that capability is impossible to emulate with disk or extended memory implementations of expanded memory.

Initialization is handled automatically by the C run-time startup, so no extra work is necessary. Termination is a bit trickier. When a program terminates and returns to DOS, DOS automatically frees up any memory used by the program and makes it available for the next one. Unfortunately, this does not happen with expanded memory. A program must always explicitly free its expanded memory pages or they will be unavailable for use by other programs until the machine is rebooted. All exit paths from the program must be covered. Even Ctrl-break must be intercepted.

Storage allocation in handle space is done by *handle_malloc( )*, *handle_realloc( )*, and related functions. Each 16K page is converted into a heap, with the usual free list data structure. There is a special data structure in page 0 that contains the size of the maximum available free block in each page, so the storage allocator can directly swap in the page it needs to allocate from, instead of sequentially swapping in each page until the allocation succeeds. This is of critical importance if a disk-swapping expanded memory emulator is used! If there is insufficient free expanded memory space to satisfy an allocation request, or if the size of a request exceeds 16K, the routines fall back to allocating from conventional DOS memory.

The Expanded Memory 3.2 subset of EM 4.0 is used to provide maximum portability; in fact, the more exotic features of 4.0 aren't needed for handle support. It's possible to use expanded memory in addition to using handles, though the two uses should be kept independent to avoid conflicts.

When to Use Handles

The overhead of dereferencing handles is much higher than the overhead of dereferencing pointers, both in program size and speed. Therefore, the best candidates for handles are those data structures that are infrequently accessed both in the number of times the dereference statically occurs in the program and in the number of times the dereferences are executed.

Expanded memory makes it possible to have two physical pages actually map onto the same logical page

Handles give a sharp increase in the amount of memory available. The proper data structure for something that will reside in handle space is one that favors speed over memory compactness. Locality of data is also important (locality means that related data should be clustered into the same page, increasing the likelihood that the desired page is already swapped in). For example, a bubble sort across data in handle space is to be avoided. If your expanded memory implementation swaps to disk, the swap file on disk may get read and written in its entirety several times during such a sort!

If database-wide searches are necessary, try putting the access structure in conventional memory and put the leaves in handles. This makes the lookup fast and efficient, and the end data is paged in only when it is needed.

Listing One (page 110) shows a version of the classic Unix wc (word count) program, converted from using the standard pointer and dynamic storage allocation to using handles and handle storage allocation. The primary advantage of using handles in a program such as wc is that they increase the program's capacity without a significant performance penalty. The handle version of wc is just as fast as the original version, which used regular pointers. Files with several megabytes of text can be processed by the handle version (assuming that expanded memory is available).

Portability is achieved simply: The *_handle* keyword is #defined to nothing, so all handles revert to being regular pointers. Second, the handle storage allocation routines can be #defined to be the standard malloc, free, and so on, routines. Thus, handle-specific code can quickly be made portable to other compilers and environments. The Zortech compiler provides a simple mechanism for eliminating handles from a program: Simply add the following line to any program before it #includes handle.h: *#define NO_HANDLE 1*

Debugging

Debugging C applications that use a lot of dynamically allocated memory is notoriously difficult. Experienced C programmers have reluctantly learned to live with this. Unfortunately, handle-based applications have all the liabilities of pointers, plus a few more. The worst problem is when a stray pointer places unwanted data into a page that has been swapped out. Such a bug will only rarely exhibit symptoms, and therefore can be difficult to pin down. Another problem occurs when more than four dereferenced handles are used simultaneously. The bug will only show up if they happen to all fall on different logical pages. Here are some techniques you can use to deal with these problems:

- Write and debug the program with handles disabled (#defined to look like ordinary far pointers). If handles are then enabled and the program fails, you can confine your search for the bug to the use of handle pointers.
- Use object-oriented style for data structures that use handles. This will confine the actual code that dereferences the handles to a few places, hence a smaller place to look for bugs.
- Carefully look for simultaneous accesses through handles. Satisfy yourself by doing "gedanken" experiments (thought experiments) that it will never be greater than four. Test suites are unreliable at flushing out these types of bugs; the code simply has to be written correctly.
- Look for function calls that could dereference handles. As always, program defensively. Assume that all function calls invalidate previous handle dereferences.

When converting code from using *malloc* to *handle_malloc*, watch out for converting from: *p = (char *) malloc(n);* to: *h = (char *) handle _malloc(n);* instead of the correct: *h = (char _handle *) handle_malloc(n);* The cast to *(char *)* dereferenced the handle and stored a far pointer into *h* instead of the handle!

(continued from page 44)

Other Uses

Handle space could be a disk file. The disk file is divided up into blocks of equal size (16K seems a good number). These are paged in and out of a small number of physical buffers (>=2). An advantage of a disk file implementation is that the limit on the size of handle space is equal to the size of the disk. Also, the data can persist from one invocation of the program to another. Disk paging would, however, be rather slow.

Extended memory can be used for handles space. Extended memory is the memory above 1 Mbyte on 286- and 386-based computers. It is only accessible via protected mode (DOS runs in real mode). To use extended memory, the CPU must switch into protected mode, copy the extended memory pages to and from real-mode buffers, and switch back to real mode. It's not as fast as expanded memory hardware, but it's much faster than paging to disk.

The minimum number of simultaneous accesses is determined by the number of physical pages available simultaneously. With expanded memory, this is four. It is recommended that at least two be available, otherwise functions such as *memcpy( )* could not be used to directly copy from one handle to another; instead, a temporary buffer in real memory would have to be used. Programs written for Microsoft Windows use a handle-like storage allocation scheme. The library implementation of the handles could be written as a shell around the Windows functions, thus easing one of the more frustrating things about programming for Windows.

Handles in VM Operating Systems

Handle pointers are useful for other applications besides extending the memory space available. Under virtual memory operating systems, such as Unix or OS/2, there is no need for more memory. There is also no hardware support for bank-switched memory like expanded memory.

Alas, one fundamental problem always remains. There is usually a difference in how a data structure is stored in memory and how it is stored on disk. Pieces of a data structure in memory are typically connected to one another by pointers. Memory pointers written to disk have no meaning when read back off a disk, so the data structure must be translated when written to disk. The memory pointers are converted into symbolic references when written out; when the data structure is read back from the disk, the symbolic references are converted back to real pointers.

Handle pointers can solve this problem. Recall that a handle consists of a logical page number and an offset into that page. The logical pages can simply be represented by disk blocks. The library implementation of handle pointers would be rewritten to read and write pages from a specified disk file rather than bank-switching expanded memory. The in-memory data structure is exactly the same as the disk file structure. Writing out the data file becomes a call to a handle function that flushes any in-memory pages to disk. Reading in the data structure becomes the simple task specifying which file to use! Dereferencing a handle will cause its page to be read in from disk. As an additional benefit, disk I/O will be minimized. Two major applications that might benefit from this approach are databases and applications that consist of multiple programs that share data via files.

Wrapping Up

Handle pointers are an elegant solution to an entire class of programming problems. The first (and what the Zortech implementation initially addresses) is extending the address space available. The second is in simplifying and speeding up writing a data structure to disk and reading it back again. The third is in defining a file format by which data can be easily read and written to files accessed by multiple programs.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

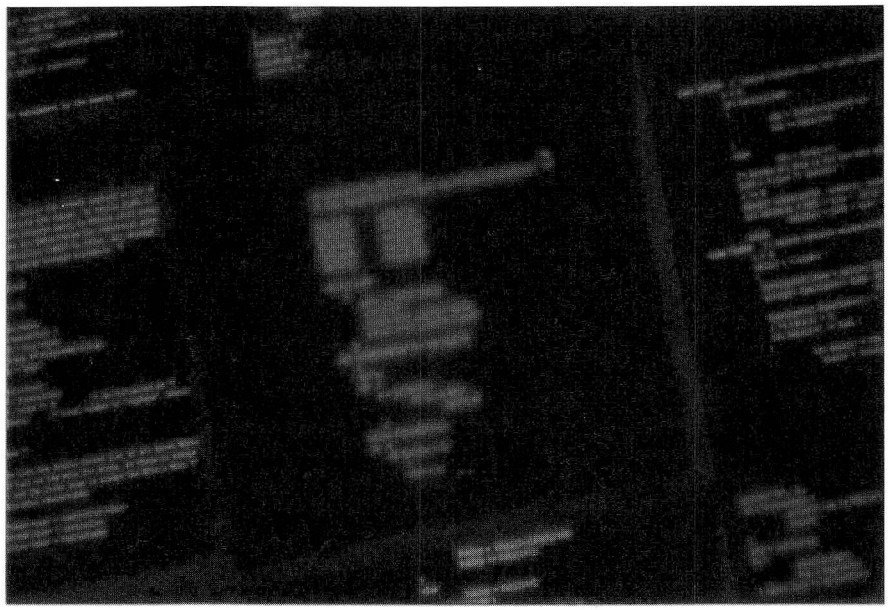
(Listing begins on page 110.)

Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

Object Swapping

Familiar memory management techniques don't always apply to OOPS

Jan Bottorff and Jim Bolland



There have been several technological stages in the development of object memory management. These have consisted of the simple memory resident strategy, the Xerox PARC OOZE and LOOM systems, and operating system-based virtual memory.

The oldest answer to the memory problem was to keep expanding memory to fit increasingly sophisticated use. Although fast and simple — and suitable to some situations — this approach tends to be expensive. We have all proved to ourselves the truth of the old programmer's axiom that the complexity of software grows to fill all available memory. Obviously, unlimited memory expansion became an unfeasible solution to the needs of users.

In the mid-seventies, a team of developers at the Xerox Palo Alto Research Center implemented the first true object-swapping system for Smalltalk-76 called OOZE.[1] Later they implemented a second system called LOOM.[2] The radical innovation found in both of these systems was that they swapped small pieces of memory, namely objects. LOOM contained an advancement over OOZE: In addition to swapping objects, the virtual address space itself was virtualized.

As time went on, machines with virtual addressing and larger and larger physical memories became common. Programs tended to use the operating system's paging instead of doing their own memory management. An object-oriented system in a paging environment causes a lot of paging activity because object access is very random. A partial solution was to keep frequently used objects together in a small set of pages. This made object access less random, thus reducing paging activity. The one drawback on some systems is that this requires hardware that supports paging.

Many newer environments have some means of providing virtual memory: OS/2 1.2 swaps segments of up to 64K-bytes while Unix and OS/2 2.0 provide paging. These virtual memory systems are an improvement over simpler systems, but they don't address some of the unique problems of an object-oriented environment.

If you analyze the access patterns of a large system written for object-oriented environments such as Actor or Smalltalk, you find the memory address space is accessed in a much more random pattern than in a C program. You also find that in each subsystem (for example, a specific type of window interaction or the processing of an algorithm) the number of objects that must be accessed is fairly small, typically a few hundred. If you combine both of these characteristics — random distribution of objects through the address space and the small working set needed for each subsystem —

Jan is a system software developer specializing in object-oriented languages and graphical user interfaces. He has been writing software and has been consulting for more than 15 years. He can be reached on CompuServe at 74775,546. Jim is a software engineer specializing in operating system internals and computer languages. He can be reached at The Whitewater Group, 600 Davis St., Evanston, IL 60201.

(continued from page 48)
the problems of conventional virtual memory systems become apparent.

Consider the following comparison among a segment swapping system, a paging system, and an object swapping system such as that in Actor, which is a pure object-oriented language and program development tool for the Microsoft Windows environment. Assume in this example, that the total universe of objects is spread over 360K of address space. Also assume there are 10K objects with an average size of 36 bytes in the complete system. Using 10 percent of the objects as the working set only consumes 36K-bytes. This large working set would actually be the combined working set of several subsystems. These represent typical values.

In a segment swapping system with 64K-byte segments, there will be about 1820 objects per segment. Due to the randomness factor, each segment would only hold about 182 objects or 6.5K-bytes for the working set.

In a paging system, there will be about 114 objects per page. Per page, about 11 objects or 396 bytes will actually be used for the working set.

In both segment swapping systems and paging systems, the entire 360K address space should be memory resident. Otherwise complete access to the working set will cause the system to swap or page constantly.

Virtual memory systems are an improvement over simpler systems, but they don't address some of the unique problems of an object-oriented environment

A better solution is to swap much smaller units of memory, specifically, individual objects. In Actor, the working set of 1000 objects would consume 36K-bytes. An additional 8K would be used by the system to maintain swapping information. The total memory required to hold our working set is only 44K-bytes. This represents only 13 percent of the memory required for a segment swapping or paging system. Once the working set is loaded, execution performance is nearly the same as other swapping alternatives.

Object Swapping

In terms of memory management, Actor objects are broken up into two major categories: static and dynamic. Dynamic objects are created by the executing program and managed by an internal garbage collection system in the Actor kernel.[3] They are typically temporary results used only while the program is executing. Static objects are created and deleted at program development time using the development system. Most static objects will be compiled program code and constants. Static objects typically outnumber dynamic ones by a ratio of 5 to 1. Their required memory also follows the 5 to 1 ratio. For these reasons, object swapping was designed to handle static objects only.

Data Structures

Internal Actor data structures allow static memory requirements to be reduced

(continued from page 50)
by about 87 percent. These data structures can also be applied to other development environments.

Every object, except for Integers and Characters, has an entry in a large array structure called the "object table" or OT (see Figure 1). The OT entry contains the current address of the object in memory or on disk if it is swapped out. It also contains two flags pertaining to the object's state. One flag is called the "touched bit," which indicates whether or not an object has been accessed. The other flag is called the "dirty bit," which indicates whether or not the object has been changed.

For objects that are swapped into memory, a header is placed at the beginning of the object. This header contains a few flag bits, the disk address where the object was swapped in from, and an age field used for discarding objects from memory. The header also contains other information used in Actor, such as the object's size.

A block of memory is allocated at program initialization time to hold all of the memory resident static objects. This is called the "swap area." A typical size for good performance is around 50K-bytes. The top 15 percent of this area is called the "overflow area." The overflow area will be explained later.

Processes Involved in Object Swapping

- Static-object creation
- Static-object referencing
- Static-object aging
- Static-object deletion
- Swap-area compaction

Static object creation occurs when the executing program explicitly requests it. An area based on the size of the new object is allocated at the current allocation pointer (see Figure 2). The disk address in the swapping header (see Figure 3) is initialized with a special signature to designate that it has no disk address yet. The age is initialized to a low value. An OT entry is allocated from a free list, and initialized with the object's memory address. New objects are marked as dirty in their OT entry. The object is now ready for use by the system.

Objects are accessed via their object pointer (OP) value. The OP is simply the offset into the OT. Object access is shown in Example 1. Addressing turns an OP into a physical address for the

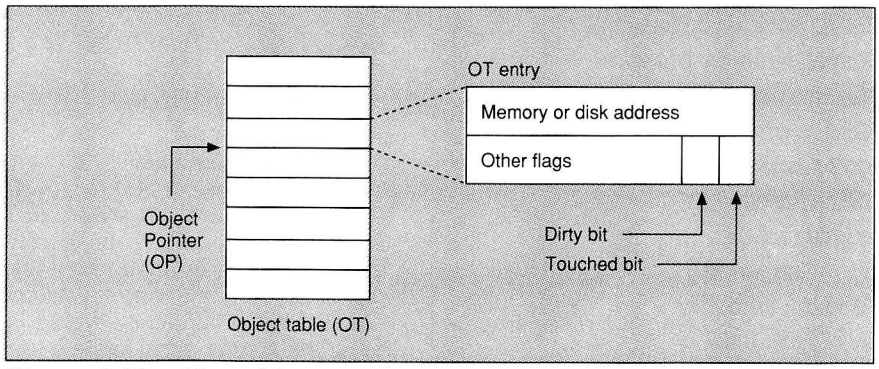


Figure 1: *The object table*

(continued from page 52)
object. This process swaps the object in from disk if necessary. An important property of addressing an object is that no other object's physical address may be changed in the process. This stability of addresses is required when the system needs to address more than one object at a time. Moving one object would invalidate the other object's address. If swapping in the addressed object requires more memory than is available in the swap area, a fatal error would occur. However, Actor is internally tuned for this not to happen under normal conditions. If an object is already in memory, its touched bit is just set in the OT and its memory address is returned.

Periodically during system execution, one of two things will happen: All needed objects are accessed or the system runs out of memory in the swap area. Before running out of memory, something must happen to free up memory. This is the job of the aging process.

As the system executes a routine is called periodically that walks through a few object table entries. When the end of the OT is reached it starts over at the beginning.

For each OT entry the touched bit is tested. If the object was touched since the last pass, the age field in the object header is set to 0, indicating the object was recently used. As the system runs, accessing objects and aging them, the age fields indicate which objects were recently used and which were not. In this way, objects can be discarded on a least recently used (LRU) basis.

When an object reaches a certain age it is tested for being changed. The dirty bit in the OT is used to determine this. If the object is unchanged, the swap file on disk still contains a correct version of the object. In this case, the OT entry is updated with the old disk address and the memory block is marked "discarded." If the object has changed, it is written back to its allocated location on disk. The aging process is shown in Example 2.

Except during periods of many object creations, objects are usually not dirty

Except during periods of many object creations, objects are usually not dirty. The typical ratio is about 500 to 1 for clean to dirty objects. This means that discarded objects rarely cause any disk activity; only addresses are moved around in memory.

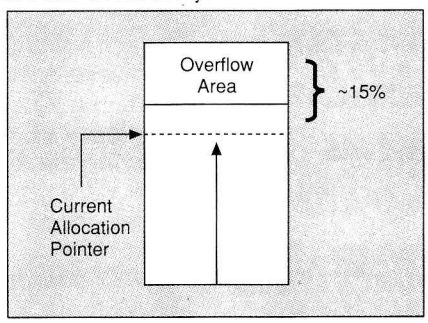


Figure 2: *Swap area*

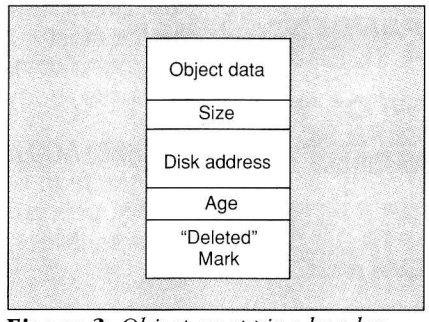


Figure 3: *Object-swapping header*

A special case occurs when a newly created object is swapped out for the first time. This is detected by a special invalid disk address. Space for the new object is allocated at the end of the object swap file and the object is written out. It is then treated like all the other objects in the system.

If an object is accessed again after being discarded but before the swap area is compacted, a new memory block is allocated and the object is swapped in. This does leave an old, unreferenced copy of the object somewhere else in the swap space, but this dual allocation of memory for a single object rarely occurs.

Swap Area Compaction is the means by which space used by discarded objects in the swap area is recovered. As objects are swapped in, the current allocation pointer into the swap area eventually reaches the beginning of the overflow area. A flag is set, indicating that swap area compaction is desired. At the moment of access, the system may be using the address of other objects, so compaction does not occur immediately. It's deferred until specific sync points are reached. Sync points are places in the execution of Actor when there are no object addresses in use, or more correctly, the addresses are at a known location. The system may keep executing and swapping objects into the swap area before a sync point is reached. This is why there is a reserved overflow area. All objects swapped in after requesting compaction but before reaching a sync point must fit in the overflow area.

When a sync point is reached, execution stops for a moment while the swap area is compacted. The compaction process updates the physical address in the OT of any static objects currently in the swap area. It does this by first overwriting the memory address in the OT with some of the data in the object header. It then overwrites the object's saved data area with the OP of the object. This allows an efficient sweep through the swap area to move all valid objects to the beginning of the swap area. During this sweep all object headers are restored from information saved in the OT entry and the OT entry is updated using our current location in the swap area for the object's address. Example 3 shows pseudocode for swap area compaction.

The final process in object swapping is the deletion of objects from the swap file. Like object creation, this is done when the executing program explicitly requests it. The equivalent of a mark and sweep garbage collection is done using a temporary disk file to store any objects to be kept. As this proceeds, the OT entry for the object is updated with the new offset into the temporary disk file. On completion, the temporary disk file becomes the new object swap file, the swap area allocation pointer is reset to the beginning of the swap area, and the system starts running again. As objects are now accessed they are swapped in from the new compacted swap file. This sweep through the swap file is much like the swap area compaction process but it is applied to the entire swap file.

To save the state of the system, a procedure known as "taking a snapshot" is used. A snapshot simply creates an image file that contains copies of the OT, the swap area, and the current swap file. To start up the Actor

```
Set the object's touched bit
if object not in memory
then
   Swap object in
   Update OT entry
endif
Get the object's address
```

Example 1: *Object access*

```
for "a few" OT entries
   if object is touched
   then
     Set object's age to 0
   else
     if object's age is too old
     then
       if object is dirty
       then
          write object to disk
       endif
       Set object address in OT to
         disk address
       Mark object memory as
        "discarded"
       else
         Increment object's age
      endif
    endif
endfor
```

Example 2: *The aging process*

system, the snapshot process is reversed: The OT and swap area are read from the image file into memory and a swap file (which is a working copy of the swap file area in the image file) is created. Dynamic objects are also stored in the snapshot file and loaded into memory when the snapshot file is restored.

Conclusions,

In this article, we have shown how to implement a virtual memory system that is optimized for object-oriented languages. It's unique design reduces memory required by approximately 87 percent, allowing much better utilization of available memory. Because special hardware support is not required, this architecture can run on many kinds of machines.

The performance delivered is usually quite high. Tests have measured object "swap in" rates as high as 320 objects per second. Average performance is about 75 to 100 objects per second. Disk caching and disk speed can affect this greatly.

New operating environments with built-in virtual memory support can still benefit from object swapping. Unless enough physical memory for all running programs exists, considerable paging activity may occur.

By using object swapping, more complex programs can be run before performance severely degrades from paging activity. A developer will be able to produce an application that runs efficiently on a larger base of machines. This also allows more sophistication to be incorporated into the application for a specific hardware configuration. Large applications can easily run on 640K machines.

Implementing an object swapping system in an application written in C can be done, but will require the programmer to explicitly call access routines whenever an object is needed. A paging system is easier for the programmer, but does not make optimal use of memory. Having the object swapping system integrated inside a language such as Actor, makes its use transparent to the programmer. This allows Actor to make optimal use of memory and free the program from having to do anything specific to support it.

```
for each valid OT Entry
  save object header info in
    OT entry
  save OP in object header
endfor
Set allocation pointer to point at
    the start of the swap area
for each object in the swap area
    if valid object
    then
      Move object to current
        allocation pointer
      Restore object header
        from the OT using saved OP
      Set OT address equal to
        allocation pointer
      Add size of object to
        allocation pointer
    else
      Skip this object
    endif
endfor
```

Example 3: *Pseudocode for swap area compaction*

References

1. Kaehler, Ted, "Virtual Memory for an Object-Oriented Language," *Byte*, August 1981.

2. Kaehler, Ted and Krasner, Glenn, *LOOM - Large Object-Oriented Memory for Smalltalk-80 Systems*, Smalltalk-80 Bits of History, Words of Advice, pp 251-270, Krasner, Glenn, editor, Addison-Wesley, 1983.

3. Duff, Charles, "Designing an Efficient Language," *Byte*, August 1986.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 4.**

A Memory Controller

Extensions to your library routines malloc *and* free

Robert A. Moeser

Many useful programs have no need for storage management. The memory they need can be calculated at compile time, and when run time comes, either enough memory is available or it isn't. Other programs have a relatively simple pattern of memory usage; they consume memory in the course of execution and never return any until terminated, when the system grabs it all back. Then there are cases of more capricious memory use — in interactive programs, for example, where a human calls the shots. You can implement programs with dynamic and unpredictable memory usage using the routines in the standard C library, but this can result in complex code that is hard to implement, read, and debug.

This article presents a set of routines for memory management. This memory control package is offered as an extension to your library routines *malloc* and *free*. It implements a free-list approach to aid in recycling portions of memory. It consists of a mere seven functions and is quite compact.

malloc and *free* Basics

Most C libraries come with two basic routines, *malloc* and *free*. *malloc* returns a pointer to a block at least the size requested. It is then up to the program to keep track of the block, using it however it will, until it is no longer needed. A call to *free* will then return that block to storage and make it available for use in subsequent requests. There are other allocation routines in the standard C library, but they are just convenience routines and can be expressed easily in terms of *malloc* plus some additional code.

Rob is a freelance programmer and can be reached at 67 Arlington Street #3, Brighton, MA 02135.



In many cases, writing a small routine to hide the call to *malloc* can make code for allocating structures a little easier to read. From Kernighan and Ritchie, for example, is the classic "tnode allocator" shown in Example 1. *talloc* hides some messy details and localizes the calls to the library storage allocator for instances of "tnodes" to one routine. These instances are freed simply by calling *free*.

A little more sophisticated is the program that anticipates a recurring need for structures of a certain type and provides not only a special allocator but also a specialized *free* routine. The routine that frees instances does not return them to the library storage allocator but instead keeps them on a free-list. A free-list is simply a singly linked list

(continued on page 61)

```
struct tnode *talloc()
{
    char *malloc();
    return ((struct tnode *) malloc(sizeof(struct tnode)));
}
```

Example 1: *The K&R "tnode" allocator*

(continued from page 58)
that chains freed blocks together. The routine that doles out new instances of the requested type first checks to see if any instances are waiting around and gets one off the free-list if it can. If it can't it goes to *malloc* for "real" storage.

With the special allocator functions in place you can get blocks of storage that are just the right size, frequently without bothering the library storage allocator. Often a program will stabilize around a certain need for storage of a given kind, and memory management calls to *malloc* will fall off nicely. Listing One (page 111) is an example of this kind of free-list allocator. The two routines use one global variable as a free-list pointer, and dispense and recover instances of a structure called a "Thing." It is a nice middle ground between relying entirely on *malloc* for allocation on the one hand and making a static declaration that 10, 50, or 500 *Thing*s will be needed on the other.

All of the code was developed and tested with Think C 4.0 for the Macintosh. It was compiled with prototypes required and full pointer type-checking. The memory control package is

MCon Programming Interface Functions

This section identifies the seven routines that constitute the memory controller program interface.

newMCon

MCon newMCon(size_t theItemSize);

newMCon is used to register an object type. The input argument is simply the size in bytes of the object to be controlled, and the return value is an MCon, or "magic key." If the package is unable to register the new object, the MCon returned will be zero. This MCon is passed to the other routines in the package that need to know which object type to operate on.

newInstanceOf

void *newInstanceOf(MCon theCon);

newInstanceOf takes an MCon as its argument and returns a pointer to a new (or recycled) instance of the object the MCon stands for. The return type is void *, so it can serve as a pointer to anything. This pointer will be zero if storage is unavailable.

disposeInstance

void disposeInstance(void *theItem);

disposeInstance takes as its argument a pointer to any instance that is no longer needed. The actual storage is not freed at this time, but is kept on a free-list for objects of the same type.

purgeMCon

void purgeMCon(MContheCon);

purgeMCon takes as its argument the MCon whose free-list should be returned to the library storage allocator by *free*.

disposeMCon

counter disposeMCon(MContheCon)

disposeMCon takes as its argument the MCon for an object type that will no longer be needed, and de-registers it. The entire free-list and the memory control package's internal representation for the object type are returned to the library storage allocator by calls to *free*. The return value is the number of "orphans" that this action creates — since a client may have active instances of the object under its control, disposing of an object's MCon can create objects which now cannot be freed. This is not strictly an error, but should be a warning sign.

purgeAllMCons

void purgeAllMCons(void);

purgeAllMCons is a convenience routine, and just purges the free-list of every object type known to the memory controller. A client might call this in a desperate attempt to satisfy a memory request.

disposeAllMCons

counter disposeAllMCons(void);

disposeAllMCons is also a convenience routine, and deregisters all known object types. This action can create orphans in the same sense as *disposeCon* above, and so returns the number. It really ought to be zero unless the client has created objects meant to endure until program termination. The memory controller also shuts itself down and frees all *malloc*'d storage. — R.A.M.

accessed by the client through just seven routines (see the accompanying text box). Listing Two (page 111), mem.h, has all the typedefs and prototypes for the package, and Listing Three (page 111), memories.c, has all the executable code. The memory control package uses only your library's *malloc* and *free* routines, so its system interface is straightforward. The package should therefore be easy to port to any up-to-date C compiler and machine.

The Programming Interface

The interface to the memory control package is simple. A client using the services provided sees it through a handful of functions. One function informs the memory controller of your intention to use objects of a certain size, referred to here as "registration." There are two routines: One for getting pointers to new instances of a registered object, and one for returning instances no longer needed. A fourth routine is available to flush the free-list for any object type (functionality not provided in the example above). There is also a routine used to "de-register" an object type, which the memory controller takes as its cue not only to flush the entity's free-list but also to free its own internal data for dealing with that object type.

The code acts as a black box, using pointers to create the desired behavior for any conceivable variety of objects and usage patterns

Finally, there are two convenience routines, one to flush the free-lists of all known object types and another that will de-register all known object types. While I haven't used these last two in my own work, they were easy to provide and could come in handy in case of a major context shift or processing phase change. You could even call them on program exit, although, because all the memory used is ultimately obtained by *malloc*, it should find its way back to the system by itself. The accompanying text box summarizes the seven functions of the programming interface.

(continued from page 62)

To use the memory controller package, include the mem.h header in each source file where its functions will be needed. Declare a variable of type MCon for each different object type. For now, think of it as a magic key. It is initialized when you register an object type with a call to *newMCon.* Usually these will be allocated statically, but the package makes no such restriction. So the following gets you a magic key for "objectA," which is some structure or union in your code:

```
MCon objectAMCon;
objectAMCon = newMCon(sizeof
                        objectA);
```

After that, you use this MCon to specify that it's an objectA you want when you call *newInstanceOf* to get a new instance. So the following will get you a pointer to an objectA-sized block:

```
objectA *anA;
anA = newInstanceOf(objectAMCon);
```

and *disposeInstance(anA);* will put that storage back onto the free-list for objectA. The other routines are just as simple to use.

Please keep in mind that just as it is bad to *free* something that was not *malloc*'d, so it is bad to *disposeInstance* of something foreign to the memory control package. And one certain way to make something foreign is to dispose of its controller with *disposeMCon.* Sometimes these orphans are what you want, but beware. I also caution against freeing an instance obtained by *newInstanceOf.* Keep direct uses of *malloc* and *free* away from uses of the memory controller.

Implementation

You might guess that an MCon is nothing more than a pointer to a structure which holds all the information that the memory controller needs in order to accomplish its tasks for a given object. This includes the size of the object and a pointer to a free-list of object-sized blocks.

The memory controller keeps all MCon objects on a doubly linked list to facilitate creating and disposing of them in response to a client's *newMCon* and *disposeMCon* requests, and to make *purgeAllMCons* and *disposeAllMCons* a matter of traversing the list and performing the requested operation on each MCon. A couple of internal routines do this list management.

Each MCon object also maintains some counters to track the use of its object type. They keep track of how many objects have been requested, how many are still "out there" somewhere under control of the client, and how many are currently on the free-list. Counters are used to validate the integrity of the internal data structures and can be used to glean useful information about usage patterns.

When a client requests registration of a new object type through the *newMCon* routine, the memory controller creates a new MCon object, fills in its fields, and links it into the master list of MCons. When the memory controller actually calls *malloc* to get storage for an object (the object's free-list being empty), it allocates some extra memory. This extra memory, hiding in front of the actual block you'll use to store your object data, is used to hold a copy of the MCon pointer that was used to create the block. The idea of hiding information in front of an allocated block is nothing new — in fact *malloc* itself typically uses some overhead storage this way in its own simpler efforts to manage memory.

In this way *disposeInstance* can figure out which MCon's free-list to use. When an object is on a free-list, this extra storage is reused to hold a pointer

(continued from page 64)
to the next free instance, making a singly linked list of free blocks of the same size.

Now the final trick that complicates the implementation somewhat: Since the memory control package has an unpredictable need for objects of a certain size (MCon objects), it uses its own routines to manage storage for them! It is said that a man who is his own lawyer has a fool for a client, but this use of the memory controller seems OK. The MCon's MCon is the *controlCon*. When the package is active, this MCon serves as the controller for instances of MCon objects. The actual object is also the head and tail of the doubly linked list of MCons. This MCon object is just a little different than all the other blocks handed out by the memory controller; it lacks the hidden information in front of the block, so it gets allocated by just calling *malloc* and is freed (if *disposeAllMCons* is called, shutting down the controller) by a call to *free*.

Figure 1 shows a diagram of how memory might look after some activity involving the memory controller has occurred.

Extending the System

The basic mechanism can be easily modified to keep more information about patterns of usage. Any amount of additional data, such as serial numbers or time stamps, can be hidden with each instance. The various mechanisms can be turned off and on in one place for development, testing, and production phases.

Additional functions would be easy to add — for example, a copy function to return a copy of an instance. Bringing such extensions into the purview of a single memory controller like this one provides one-stop shopping for all

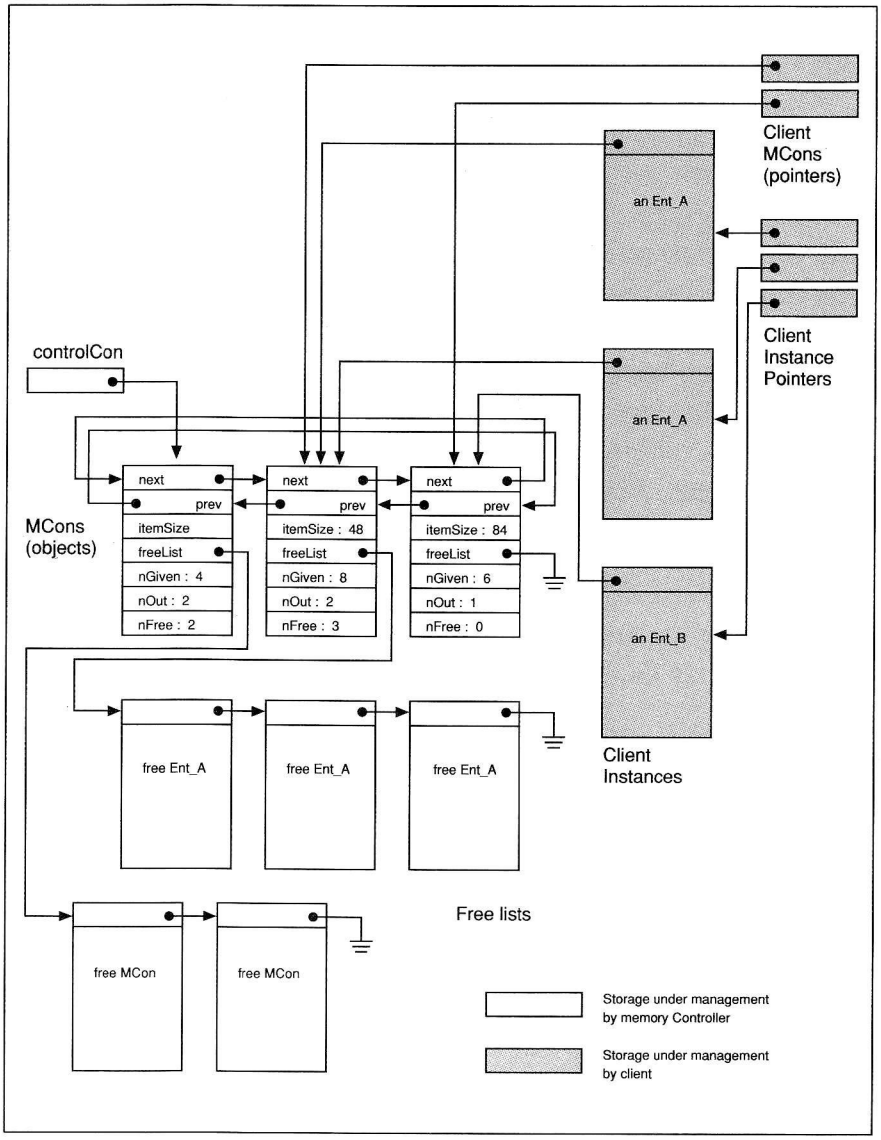


Figure 1: *Memory controller map*

(continued from page 66)
kinds of tricks.

Variable-sized objects such as strings present a problem in this scheme. In my own code, I once used a method I dubbed the "Procrustean Allocator." I had MCons for a few different object sizes and rudely grabbed the nearest one that fit. The sizes were all powers of two, and I used the services of the memory controller to accomplish the allocation task with a little array of MCons. There is, of course, no reason you can't continue to use *malloc* and *free* wherever you like, as the memory controller in no way precludes other uses of these basic functions.

Listings Four and Five (page 112) provide a complete program that torture-tests the memory controller. It randomly registers objects of various sizes, randomly creates and frees instances of them, and occasionally de-registers object types.

Conclusion

This set of routines provides simple access to a particular memory management technique. The code acts as a black box, using pointers to create the desired behavior for any conceivable variety of objects and usage patterns. It is self-starting and self-terminating and provides its services with minimal overhead. By using this package, you can improve the readability, if not the performance, of programs that call it. With a few additions, it could serve as a statistics-gathering device for memory use or as the basis for writing in a traditional language, using an object-oriented philosophy.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 111.)

Vote for your favorite feature/article.
Circle Reader Service **No. 5.**

Demystifying 16-Bit VGA

There's more to VGA than meets the eye

Michael Abrash

A year or two ago, a friend in the industry made the mistake of mentioning to a headhunter (excuse me, an employment recruiter) that he was interested in hiring someone with object-oriented programming skills, which were rare at the time. From that moment forth, every résumé that came through that particular agency boasted of object-oriented programming experience. My friend would ask each candidate if he or she had any experience with object-oriented programming, and each one would answer yes. Then my friend would ask exactly what object-oriented programming is. Not a one of them had a clue.

Which brings us, in a slightly roundabout way, to 16-bit VGA.

What 16-Bit VGA Really Is

16-bit VGA. Those seductive words, promising what every PC user craves — performance — are everywhere. "16-bit VGA," ads shout. "16-bit VGA: Does It Matter?" articles and reviews ask. "Do I need a 16-bit VGA?" every power user wonders, and, "Can I afford one?"

It seems as if 16-bit VGAs have been with us forever, or at least since IBM came down from the mount with the PC, but in fact they're a relatively new development, dating back only a year or so. (IBM's Display Adapter, the PC-bus version of the VGA, was — and still is — an 8-bit adapter.) As such, they're a lot like object-oriented programming was when my friend was looking to hire a programmer: Widely used as a buzzword, claimed by many, and not particularly well understood by the reviewers who write about them, the users who buy them, or the developers who must deal with them. Most significantly, 16-bit VGA isn't a standard, but rather a catchall name for a variety of VGA enhancements. Any VGA with a 16-bit bus interface (that is, with two connectors that plug into the AT bus) is bound to be advertised as a 16-bit VGA, but all 16-bit VGAs are not created equal, and the value of a given 16-bit VGA varies greatly depending both on the sorts of 16-bit operations it offers and on how you use it.

DDJ's readers, who are both developers and users (and sometimes reviewers as well) would surely benefit from a solid understanding of what 16-bit VGA really means and what benefits the various types of 16-bit VGA offer. It's from that perspective that I'll attempt to clear up some misconceptions and confusion about 16-bit VGA in this article; most importantly, we'll see why (happily and contrary to some reports) programmers need not treat 8- and 16-bit VGAs differently.

Michael works on high-performance graphics software at Metagraphics in Scotts Valley, Calif. He is also the author of Zen Assembly Language *published by Scott, Foresman & Co., and* Power Graphics Programming, *from Que.*

Performance

Let's begin by placing the one and only reason for the existence of 16-bit VGAs — performance — in context. 16-bit VGAs can allow screen-oriented programs to run faster than other VGAs, but it's not really correct to say that they run those programs faster; more accurately, 16-bit VGAs slow programs down less.

What's the distinction? More powerful graphics adapters, such as the 8514/A and adapters built around the TI 34010 graphics chip, have dedicated processors and specialized hardware that allow them to offload work the CPU would otherwise have to do and to perform that work very rapidly, so they really can run screen-oriented programs faster than if the CPU were required to do all the work itself.

In contrast, the best a VGA can do is get in the processor's way less. You see, all VGA-based graphics operations are performed directly by the CPU — the 8088, 80286, 80386, or whatever processor happens to be in a given PC. The VGA has only a bit of hardware assist on board, and has no independent processing ability at all. The VGA is basically a set of I/O ports and a memory map to be manipulated directly, and at a very low level, by the CPU. Given that, the only way a VGA can contribute to improved performance is by not slowing the CPU, that is, by allowing the CPU rapid access to I/O ports and memory. Ideally, the CPU would be able to make every access to VGA memory as rapidly as to system memory, and likewise for I/O ports.

That's the ideal, but it's far from the reality. To understand why, we must first understand how the AT bus handles 8-bit adapters. That discussion has two facets: The splitting up of 16-bit

accesses to 8-bit adapters, and the automatic slowing down of all accesses to 8-bit adapters. Before we can cover those topics, however, we must talk about wait states.

Wait States in the AT

Wait states are cycles during which the CPU does nothing because the bus or some memory or I/O device tells it to wait. Put another way, they're states that are thrown away by the CPU at the request of external circuitry. While wait states aren't desirable because they reduce performance, they're necessary because they allow slower memory and I/O devices to function properly with a fast CPU. For example, the 80286 is capable of performing a memory or I/O access in just two cycles. However, the bus inserts one-wait-state on each access to most 16-bit devices, including system memory, in a standard AT, as shown in Figure 1. This increases access time to three cycles, reducing overall performance but allowing the use of slower, cheaper chips.

Wait states are also inserted by an adapter whenever the adapter can't respond at the maximum speed of the bus or processor. As we'll see, some VGAs insert additional wait states, while others avoid additional wait states at least some of the time.

16-Bit Accesses to 8-Bit Adapters

There are two fundamental classes of adapters that may be plugged into the AT bus: 8-bit adapters and 16-bit adapters. The two are distinguished by the extra bus connector that appears only on 16-bit adapters; in addition, 16-bit adapters must announce to the bus that they are indeed capable of handling 16-bit accesses, by raising a particular bus line on the 16-bit connector early on during each access.

What happens if an adapter doesn't have the 16-bit connector, or if it doesn't announce that it's a 16-bit device? Why, then the AT's bus does two things. First, the bus splits each word-sized access to that adapter into 2-byte-sized accesses, sending the adapter first 1-byte and then the other. That's not all the bus does, though: During each of those byte-sized accesses to an 8-bit

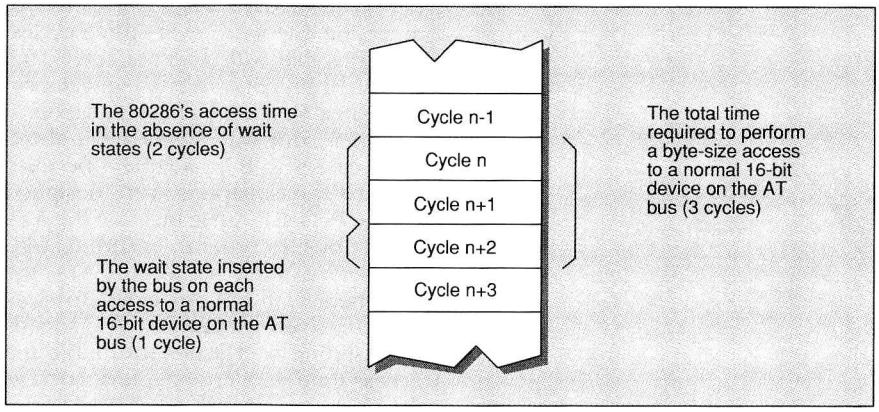


Figure 1: *At least three cycles are required for each byte-sized access (or word-sized access to an even address) to a normal 16-bit device on the AT bus. Each access to a special zero-wait-state, 16-bit device may take as few as two cycles; each word-sized access to an odd address of a 16-bit device takes twice as long as a byte-sized access*

adapter, the AT bus inserts three extra wait states (in addition to the one-wait state that's routinely inserted), effectively doubling the access time per byte of such adapters to six cycles, as shown in Figure 2. These extra wait states, which I'll refer to as 8-bit-device wait states, form a pivotal and little-understood element of 16-bit VGA. Together with the splitting of word-sized accesses into 2-byte-sized accesses, 8-bit-device wait states can quadruple the access time per word of 8-bit adapters; instead of accessing one word every three cycles, as is possible with 16-bit adapters, the AT can access only 1-byte every six cycles when working with 8-bit adapters.

Three extra wait states are inserted on accesses to 8-bit adapters because the first 8-bit adapters were designed for the PC's 4.77-MHz bus, not the AT's 8-MHz bus. In order to ensure that PC adapters worked reliably in ATs, the designers of the AT decided to slow accesses to 8-bit adapters to PC speeds by inserting wait states to double the access time. Modern adapters, such as the VGA, can easily be designed to run at AT speeds or faster, whether they're 8- or 16-bit devices — but the AT bus has no way of knowing this, and insists on slowing them down — just in case. It should be obvious that true 16-bit operation, where an adapter responds as a 16-bit device and handles a word at a time, is most desirable. Not at all obvious is that it's also desirable, that an adapter respond as a 16-bit device even if it can internally handle only a byte at a time. In this mode, an inherently 8-bit adapter announces to the bus that it's a 16-bit device; on writes, it accepts a word from the bus and then performs two 8-bit writes internally, and on reads, it performs two 8-bit reads internally and then sends a word to the bus. From the perspective of the bus, each word-sized operation seems to be a 16-bit operation to a true 16-bit adapter, but in truth two accesses are performed, so the operation takes twice as long as if the adapter were a 16-bit device internally.

Why bother? The advantage of having an 8-bit adapter respond as if it were a 16-bit adapter is this: The bus is fooled into thinking the adapter is a 16-bit device, so it doesn't assume that the adapter must run at PC speeds and doesn't insert three extra wait states per byte. From now on, I'll use the word "emulated" to describe the mode of operation in which an adapter that's internally an 8-bit device responds as a 16-bit adapter; this mode contrasts with the true 16-bit operation offered by adapters that not only respond as 16-bit devices but are 16-bit devices internally. AT plug-in memory adapters, for example, are true 16-bit adapters. 16-bit VGAs, on the other hand, may be either true or emulated 16-bit adapters; in fact, as we'll see, a single VGA may operate as either one, depending on the mode it's in.

Emulated 16-bit operation is at heart nothing more than a means of announcing to the AT bus that an inherently 8-bit adapter can run at AT speeds thereby making the three 8-bit-device wait states vanish. While emulated 16-bit adapters can run up to twice as slowly as true 16-bit adapters (word-sized accesses must still be performed a byte at a time), emulated 16-bit operations can double the performance of an inherently 8-bit adapter that is otherwise capable of responding instantly, by cutting access time from six to three cycles.

8-Bit and 16-Bit Adapters Don't Mix

If there is one 8-bit display adapter in an AT, all display adapters in that AT must be 8-bit devices. Consequently, all 16-bit VGAs automatically convert

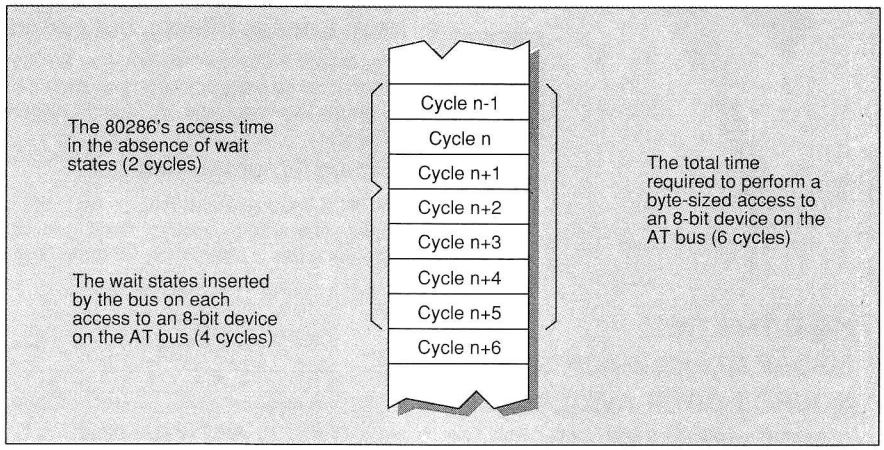


Figure 2: *At least six cycles are required for each byte-sized access to an 8-bit device on the AT bus*

(continued from page 72)
to 8-bit operation if an 8-bit adapter is present. If you put a monochrome adapter in your AT along with your expensive 16-bit VGA, what you'll get is 8-bit VGA performance.

Why this happens is a function of the addressing information available to an adapter at the time it has to announce it is a 16-bit device; I lack both the expertise and the space to explain it in detail. The phenomenon does exist, however, and the conclusion is simple: Don't bother getting a 16-bit VGA if you're going to put an 8-bit display adapter in your system as well.

Wait States in Other AT-Bus Computers

All AT-bus 80386-based computers slow down both 8- and 16-bit adapters considerably. (Obviously, 16-bit VGAs are wasted in 8-bit PCs, in which they operate as 8-bit devices.) AT-bus 80386 computers insert wait states — often a great many wait states — on accesses to 16-bit devices in order to slow the bus down to approximately the 375 nanoseconds excess time speed of the AT bus, so that AT plug-in adapters will work reliably. A 33-MHz 80386 is capable of accessing memory once every 60 nanoseconds (two cycles); ten wait states must be inserted to slow accesses down to about the 375-nanosecond access time of a standard AT. Clearly, memory on 16-bit plug-in adapters responds considerably more slowly than 32-bit memory in 80386 computers; the 80386 in the above example is idle more than 80 percent of the time when accessing plug-in 16-bit memory. Because of this, you can expect to see VGAs built onto the motherboards of most high-performance computers in the future, thereby completely bypassing the many wait states inserted by the AT bus.

Most significantly, 16-bit VGA isn't a standard but rather a catchall name for a variety of VGA enhancements

In many 80386 computers, 8-bit adapters are worse still. A number of 80386 motherboards slow accesses to 8-bit adapters down to about the PC's bus speed of 838 nanoseconds per access, which could mean as many as about 25 wait states in the above example. However, a number of 80386 computers slow both 8- and 16-bit adapters down to AT speeds; in these computers, the performance distinction between 8- and 16-bit adapters vanishes.

What of Micro Channel computers? They don't distinguish between 8- and 16-bit devices — but that's a moot point, because Micro Channel computers have VGAs built right onto the motherboard. For the remainder of this article, I'll talk only about AT-bus VGAs.

To summarize, byte-sized accesses to an 8-bit adapter take twice as long on ATs and many 80386 computers as accesses to a 16-bit adapter if both adapters are otherwise capable of responding instantly; word-sized accesses take twice as long again on 8-bit adapters. Most VGAs aren't capable of responding instantly, though, and in memory and I/O response time lies another part of the 16-bit VGA performance tale.

VGA Memory

Before we can look at the response time of VGA memory, we must clarify exactly what sort of VGA memory we're talking about. For practical purposes, there are three types of VGA memory: ROM, text-mode memory, and graphics-mode memory. A 16-bit VGA might actually provide 16-bit access in one, two, or all three areas, and 16-bit access might be either true or emulated for ROM and text-mode memory. In addition, 16-bit access provides different benefits in each area. Next, we'll look at 16-bit operation in each VGA memory area, and 16-bit I/O, as well.

ROM

It's easy to provide true 16-bit access to a VGA's ROM, and most 16-bit VGAs do so. If a 16-bit VGA has two ROM chips, it's a pretty safe bet that it offers true 16-bit ROM operation, which in turn translates into a performance improvement of close to four times for VGA ROM code. Just what does that massive speedup do for us?

The VGA's ROM contains an extended video BIOS that's responsible for the text placed on the screen by DOS and BIOS functions, a category that includes the DOS prompt, directory listings, and text drawn with *printf* and *writeln*, but not the text drawn by virtually any major word processor, text

(continued from page 74)
editor, or other program that offers a full-screen interface. The VGA's BIOS also provides functions for drawing and reading dots in graphics mode at relatively low speeds; again, most graphics programs ignore these functions and access the video hardware directly. Finally, the VGA's BIOS supports miscellaneous functions such as setting the color palette and returning configuration information.

Consequently, the primary benefit of 16-bit ROM access is speeding up directory listings, the TYPE command, program output sent to the standard output device, and the like, which can make the computer feel sprightlier and more responsive. On the other hand, most manufacturers provide RAM-loadable BIOSes, which are generally faster than even 16-bit ROM BIOSes, especially in fast 80386 computers, because they run from system memory. Also, many computers can copy the VGA's BIOS into shadow RAM, non-DOS system memory reserved especially for the purpose of replacing ROM with fast RAM.

On balance, 16-bit access to the VGA's ROM BIOS is nice to have when a RAM or shadow RAM BIOS is not in use, because it speeds up certain common operations. 16-bit ROM access does not, however, affect the performance of programs that do direct screen output, including most commercial PC software.

A Brief Aside on Benchmarks

It's not easy to benchmark VGAs in ways that correspond to meaningful user-performance improvements. A number of programs used to test VGAs are actually BIOS tests, because it's easy to exercise the various BIOS functions; alas, that falls far short of fully exercising the VGA standard, for many programs ignore the BIOS altogether and access the VGA's hardware directly. Other benchmarks just measure raw memory access speed; they measure ideal conditions that are rarely achieved in the real world. Raw memory access speed doesn't necessarily map well to the sorts of operations — bitblts, line draws, scrolling, fills, and the like — that performance-sensitive screen-oriented programs actually perform.

There are two types of meaningful benchmarks. Benchmarks that measure actual programs doing useful, time-consuming work (redrawing screens in Autocad or scrolling through a document in a Windows-based word processor, for example) are certainly relevant. Better yet are benchmarks you perform yourself with the VGA software you plan to use. Performance is not absolute: It is a relative measure that is meaningful only in context. What good will it do you to buy the VGA that runs Autocad fastest if you do all your drawing work in Fastcad? There's no guarantee that a VGA that performs well with one program will perform well with another, if for no other reason than that the performance of driver-based programs varies as much with driver quality as with hardware speed, and each VGA manufacturer provides its own set of drivers.

Use your own software to test drive any VGA you're considering buying. You'll be glad you did.

The VGA has up to 152K of graphics memory per screen, and needs to scan about nine million bytes of video data onto the screen every second

Text-Mode Memory

The second type of VGA memory is text-mode memory. Some VGAs provide true 16-bit access to text-mode memory, while others provide emulated 16-bit access. True 16-bit VGA is clearly the superior of the two in this context; word-sized writes actually happen quite often in text mode, as both a character and its attribute are frequently written to memory with a single instruction. Emulated 16-bit access alone may or may not improve performance, depending on whether or not a given VGA can respond fast enough to take advantage of the three cycles per byte-sized access that 16-bit emulation saves. The end result is that emulated 16-bit text-mode memory can as much as double the speed of access to text-mode memory, while true 16-bit text-mode memory improves performance by up to four times over 8-bit VGA.

A two- to four-fold increase in text-mode memory access certainly makes for snappy response for text-mode programs that go directly to display memory, including most spreadsheets and word processors. However, because relatively few bytes of display memory control an entire screen, display memory access speed is rarely the primary limiting factor in the performance of text-mode programs, so the perceptible effect of 16-bit text-mode VGA is not as dramatic as the numbers might suggest. All in all, 16-bit access to text-mode memory, such as 16-bit access to the BIOS ROM, can make for an enjoyable, if not stunning, increase in the responsiveness of certain programs.

Graphics-Mode Memory

That leaves us with just one VGA memory area to check out graphics-mode memory. While this is the area in which 16-bit VGA can do the least to increase memory access speed, because true 16-bit accesses can't be supported within the VGA standard, it is ironically also the area that brings out the best in 16-bit VGA — given the right circumstances.

As any VGA user knows, it's in graphics mode that the VGA feels slowest; that's becoming all the more apparent with the rise of graphical interfaces such as Windows and Presentation Manager. Making a non-VGA- compatible display adapter that supports faster graphics than a VGA is not difficult at all; the aforementioned 8514/A and 34010-based adapters fit that description, for example. The trick is to improve graphics performance without losing VGA compatibility, so that the improved performance automatically benefits every one of the hundreds of programs that support the IBM VGA.

Display Memory Access Speed in Graphics Mode

As I mentioned earlier, the best a VGA can do is get out of the way of the CPU to the greatest possible extent. To see why this is most important in graphics mode, let's look at a few numbers. The VGA has up to 152K of graphics memory per screen, and needs to scan about nine million bytes of video data onto the screen every second. That alone takes close to 80 percent of all available memory accesses on a standard VGA. VGA memory must also be accessed many times by the CPU in order to draw any sizable image, because there are so many pixels on the screen, in so many colors; however, those accesses must be shoehorned between the video data reads described earlier. (For further information about the conflicting demands on display memory, see my article "Display Adapter Bottleneck," *PC Tech Journal*, January 1987.)

How to resolve these heavy dual demands on display memory? One choice is to give priority to the video data and make the CPU wait frequently. This is simple and inexpensive to implement;

(continued from page 76)
the only drawback is that CPU performance suffers.

There are a variety of other approaches to VGA graphics-mode memory design, all of which improve performance to some degree. Some VGAs use faster memory than IBM's VGA does, freeing up more display memory accesses for the CPU. Other VGAs use different memory architectures, such as paged-mode or video RAM, that reduce the overhead of supplying video data and allow the VGA to service the CPU faster. There are a number of other performance-enhancing techniques in use; the point is that there are a variety of means by which fully IBM-compatible VGAs can insert fewer wait states and slow the CPU less, improving overall graphics performance.

Interesting, but what does it have to do with 16-bit VGA? Simply this: Only emulated 16-bit VGA can be implemented in graphics mode; true 16-bit VGA is a physical impossibility, as we'll see shortly. Emulated 16-bit VGA matters only because it eliminates wait states; three wait states per byte-sized access to display memory on an AT and often more in an 80386-based machine. If a VGA inserts more than three wait states per access anyway, because the memory is inherently slow or because the CPU must wait while video data is fetched, then emulated 16-bit VGA won't make a blessed bit of difference. If, on the other hand, a VGA is inherently capable of responding as quickly as normal AT system memory (as is theoretically the case with a VGA built around 120-nanoseconds VRAM), then the 8-bit-device wait states spell the difference between a VGA that responds in three cycles and one that responds in six cycles.

In a nutshell, the faster a VGA's memory architecture, the more the 16-bit interface matters. The 16-bit interface allows VGAs with inherently fast memory access times to respond up to twice as fast as they otherwise would, slowing the CPU less and allowing higher graphics performance overall.

Of course, not all 16-bit VGAs are twice as fast as 8-bit VGAs; for instance, VGAs that provide slow memory access won't benefit from the 16-bit interface at all. In addition, the overall performance improvement experienced by graphics software on even the fastest 16-bit VGA depends on the frequency with which that software accesses display memory. Plotting software that performs several floating-point calculations for each point drawn is not going to be measurably affected by VGA speed, while software that spends most of its time copying blocks of display memory around (scrolling or updating large areas of the screen, for example) may indeed run nearly twice as fast on a 16-bit VGA as on an equivalent 8-bit VGA, and the advantage over slower VGAs may be greater still. Drivers weigh heavily into the performance equation as well, as noted above.

The Myth of 16-Bit Operations

There's a myth that 16-bit VGA improves graphics performance only when 16-bit accesses to memory are used; on the basis of this myth, many people have concluded that because graphics software written for the IBM VGA generally performs 8-bit operations, it won't benefit from 16-bit VGA. Not true. As we've just seen, in graphics mode the great virtue of 16-bit VGA has nothing to do with 16-bit operations; rather, it is that the 16-bit interface serves to fool the AT bus into not inserting 8-bit-device wait states.

In fact, 16-bit VGAs must operate as 8-bit devices internally in graphics mode, offering emulated but not true 16-bit operation. This is unavoidable because the VGA architecture is an 8-bit architecture, with 8-bit internal

(continued from page 78)
latches, 8-bit data masks, and so on. While VGA designers could certainly create 16-bit latches and the like, standard VGA software, which expects the normal 8-bit setup, wouldn't work properly anymore — and running standard VGA software faster is the object of 16-bit VGA. Consequently, 16-bit VGAs actually break each 16-bit access into two 8-bit accesses internally, just as the AT bus does during 16-bit accesses to 8-bit devices, but without the three-wait states the AT bus inserts. (As I noted earlier, some VGAs really do support single 16-bit accesses in text mode; this is possible because in text mode display memory appears to the CPU to be a single, linear plane of memory, and none of the VGA's 8-bit hardware assist features come into play.)

What does all this mean? It means that programmers need not worry about altering or fine-tuning graphics code for 16-bit VGAs; standard VGA code will run fine (but faster), and 16-bit operations are no more desirable on 16-bit VGAs than on 8-bit VGAs. Hallelujah!

I/O Access Speed

Finally, we come to the last aspect of 16-bit VGA performance: I/O. I/O to the VGA ports is performed frequently in graphics mode in order to set the bit mask, the map mask, the set/reset color, and so on. These I/O accesses are subject to the same 8-bit device wait states as memory accesses, so it's desirable that VGAs respond as 16-bit devices to I/O as well as memory accesses. I/O is less critical in text mode, where it is used primarily to move the cursor, but 16-bit I/O can help there, too.

As it happens, not all 16-bit VGAs do support 16-bit I/O, so this is yet another area in which 16-bit VGAs can differ widely, and yet another feature for a VGA purchaser to check out.

Conclusion

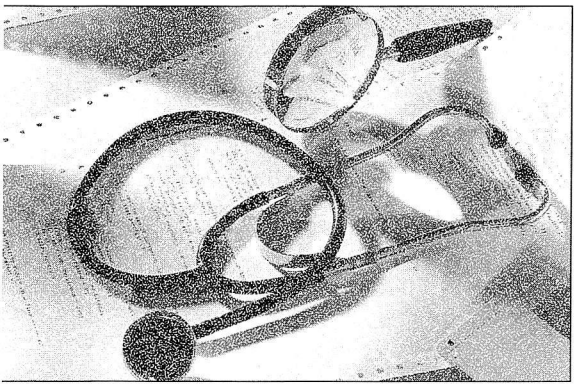
What's the bottom line on 16-bit VGA? First and most important, 16-bit VGA is a user issue, not a programming issue; developers need not spend time worrying about separate drivers or code optimized for 16-bit VGAs. 16-bit VGAs may have extended modes or special features that require or benefit from custom code, but that has nothing to do with 16-bit VGA itself, which is primarily a way to trick the AT bus into not inserting wait states, and sometimes a way to provide true 16-bit access to ROM and text-mode memory, as well.

Second, while 16-bit VGA can make text-mode operation more responsive, it produces the most visible and sorely needed improvement in graphics mode, but only for VGAs that provide memory-access times close to that of system memory. In those cases, however, 16-bit VGA can provide an appreciable performance boost, as much as doubling the execution speed of graphics software over 8-bit VGA, although the improvement depends heavily on the frequency with which the software accesses display memory and the VGA's I/O ports.

In summary, the speedup from 16-bit VGA is incremental, not revolutionary, but is significant nonetheless. The VGA is the last gasp of directly CPU-controlled, bit-mapped graphics, and 16-bit VGA squeezes the last ounce of performance from that old standard. Whether you need that extra edge depends on the software you use, but at least now you understand the many facets of 16-bit VGA and can better match your needs to the features of the many 16-bit VGAs on the market.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 6.**



Multiprocessing with Smalltalk/V

A look at the CX Multiprocessing Extension Kit

Kenneth E. Ayers

The origins of object-oriented programming in general and Smalltalk in particular are closely tied to the simulation of real-world events and systems. A primary reason for this association is Smalltalk's facility for describing the behavior of a simulated system in terms of those real objects with which the implementor is familiar. Thus, the programmer of a highway traffic simulation can create car objects and truck objects and give them behaviors that mimic their real-world counterparts. Likewise, someone who is studying the flow of people through supermarket checkout lines can create objects representing customers, checkers and baggers and have them carry out actions that are familiar to anyone who has spent time waiting behind a shopping cart.

As an added benefit, extensible systems such as Smalltalk permit the developer of a simulation system to construct his or her own task language with which to describe the flow of information back and forth between the various participants in the simulation. This task language then becomes a part of the development system.

Until recently, though, the availability of object-oriented languages such as Smalltalk were limited to those using powerful (and expensive) workstations or minicomputers. Fortunately, that changed when Digitalk (Los Angeles, Calif.) introduced Smalltalk/V and its more powerful brother, Smalltalk/V 286. And, with a growing community of devoted followers, third-party support has finally begun to emerge. Because Smalltalk is such an open-ended system, this support commonly assumes the form of toolkits that extend the capabilities of Smalltalk's built-in environment. One such toolkit is the CX Multiprocessing Extension Kit from Computas Expert Systems.

The Goods

The CX Multiprocessing Extension Kit (the Kit) provides many useful extensions to the Smalltalk/V or Smalltalk/V 286 environments. Within its 15 separate modules, the Kit provides extended functionality ranging from basic utility methods all the way up to a complete data acquisition class hierarchy. As is expected in the Smalltalk world, source code is provided for all of the classes and methods; and *filein*'s (scripts to read source code into the Smalltalk system) are available for installing each of the modules. Briefly, the modules include:

- **Basic Extensions** — are general methods used by other modules in the Kit
- **Process Extensions** — support the creation of named processes, binding variables to a process, and assigning process priorities
- **Semaphore Extensions** — provide enhancements to inter-process communications including named semaphores, limited-capacity semaphores, and synchronized message queues
- **Screen Controller** — allow updates to partially obscured windows and displaying variable graphic information over a constant background image
- **Process Status Window** — implements a standard window for real-time monitoring processes and semaphores
- **Date/TimeServices** — enhanceSmalltalk/V with the ability to format date and time output, and convert between formats
- **Event Queue Mechanism** — adds timed events (either relative or absolute date/time) and a monitor process that watches the system's clock for event triggers
- **Acknowledge Mechanism** — permits a background process to temporarily gain control of the user interface for the purpose of acknowledging or verifying actions
- **Data Acquisition Protocol** — implements a class hierarchy, under the abstract class *Instrumentation*, that provides a complete Smalltalk/V interface to the MicroMac 4000 controller from Analog Devices. The protocol, which serves as a functional template, can be modified to support other types of instrumentation controllers and subsystems
- **Miscellaneous Extensions** — contain six additional modules not directly related to multiprocessing including a

Ken is a software engineer currently employed by Industrial Data Technologies in Westerville, Ohio, where he is involved in the design of industrial graphic workstations. He also works part time as a consultant, specializing in prototyping custom software systems and applications. He can be contacted at 7825 Larchwood Street, Dublin, OH 43017.

(continued from page 82)
Form Inspector that displays the contents of a displayable image; enhancements to Smalltalk/V's Freehand Drawing (that is, painting) application and its associated Bit Editor; improvements in the behavior of the built-in *TextEditor* class; new versions of the class Browsers; and an Application Browser

The Goodies

Several of the extensions provided by the CX Multiprocessing Extension Kit depend upon capabilities added by one or more of the Goodies disks available (as options) from Digitalk. These dependencies, as well as the dependencies within the Kit's internal modules, are clearly specified in the documentation.

Of primary concern, however, is the lack of support for multiprocessing in the original Smalltalk/V. Users of Smalltalk/V must install the Goodies #1 package before any of the CX multiprocessing extensions can be used. Users of Smalltalk/V 286 do not have this concern.

Otherwise, the Data Acquisition Module depends upon the Digitalk Communication Kit (for serial communications); the Freehand Drawing extensions depend upon Goodies #1 (to load and store images); the Text Editor extensions require Goodies #2; and the Browser and Application Browser modules need both Goodies #2 and Goodies #3.

One further note must be offered because it presents a problem when attempting to load the CX Multiprocessing Kit. Smalltalk/V and Smalltalk/V 286 do not support floating point numbers without a math coprocessor present in the system. The CX Data Acquisition module is the only module that appears to require floating point capabilities. In order to load any other modules from the kit on a system that does not have an 80x87, the methods *containsFloat* and *asFloat*, in the Basic Extensions module (CXBSCEXT.PRJ), must be commented out. (Note: The Goodies #2 kit provides software floating point support.)

The Package

The evaluation copy I received was marked Version 1.0. It was supplied on a single 3.5-inch (720K format) diskette (a 5.25-inch format is also available). The data necessary to build the installation package has been compressed into several special .EXE files. One of two batch files (one for Smalltalk / V and one for Smalltalk / V 286) are invoked to initiate the creation of more than 60 source code and example files.

A concise, well written, 105-page manual describes each of the modules and provides simple installation instructions in the required sequence. The number of examples given in the manual is somewhat limited, although the Smalltalk/V source code for many others is available in various files. Consistent with Smalltalk/V's tutorial, these examples can be selected and executed using the File Browser.

Multiprocessing?

The Smalltalk/V 286 Virtual Machine (its kernel) contains support for scheduling the execution of multiple processes with consideration for priority levels. In the context of Smalltalk, a process is a block of code that is capable of executing as an independent program. During its lifetime, a process can assume any one of several states: Active, the process is currently executing and "owns" the computer; Ready, the process is ready to run but is waiting to be scheduled for execution; Blocked, the process is waiting for some resource to become available or for some event to occur (for example, a signal from another process); and Dead, the code associated with the process has reached its logical point of termination.

A process is created by sending the

(continued from page 84)
message fork or *forkAt:aPriority* to a block. An example is *[self run] forkAt:2.* In response, the Smalltalk/V kernel will create a *Process* object, whose code is the expression in the block ('self run'), and schedule it for execution at the next available opportunity. That opportunity comes when:

1. The current active process terminates, becomes blocked or voluntarily relinquishes control by executing the statement *Processor yield.*
2. There are no other ready processes at a higher-priority level.
3. There are no other ready processes at the same priority level that have been ready for a longer period of time.

Note that Smalltalk's scheduling is non-preemptive or cooperative. There is no time slicer that periodically says "Okay process, you've had enough time. Time to let this other process have its chance." In Smalltalk, it's up to the programmer to assure that processes are "courteous" and periodically give others a shot at running the show.

By its very nature, multiprocessing capabilities are ideally suited to tasks such as industrial control and monitoring, and to a class of applications known as discrete event simulations. In both of these types of applications, multiprocessing allows each individual object participating in the application to be represented by separate processes. Processes communicate with one another by sending messages (via message queues) or by signaling (via semaphores) the occurrence of some event required for coordinating their activities.

Lining Up an Example Application

To demonstrate the use of multiprocessing and apply some of the many features offered by the CX Multiprocessing Kit, I have included a simple application that provides an animated simulation of checkout counters at a typical supermarket. In this application, there are five basic classes of objects and processes:

There seems to be little evidence of redundancy or overkill

1. Customer processes (instances of class *Customer*) enter a checkout line, wait to be served, empty their carts, and then leave.
2. Checkout clerk processes (class *Checker*) accept items from the customer and hand them off to be put into grocery sacks.
3. Bagger processes (class *Bagger*) accept items from the checkout clerk and place them into grocery sacks.
4. Checkout counter processes (instances of class *CheckoutCounter*) are assigned one checkout clerk, one bagger, and a queue of customers who are waiting to be serviced.
5. The "store" itself (represented by the class *MarketSimulator*), is a single process that generates new customers and collects information about the simulation as it runs. The classes *Customer*, *Checker*, *Bagger*, and *CheckoutCounter* are all subclasses of the class *MarketActor.* Listing One (page 114) lists the classes while Listing Two (page 118) lists the simulation program itself.

MarketActor is an abstract superclass that provides all of the common functionality such as displaying and animating, accessing size and position information, and starting and stopping the object's associated process. The subclasses add additional behaviors that are more appropriate to the particular type of simulated actor.

Internally, the coordination between the checkout counter, the customer, the checker, and the bagger are carried out by passing command messages to other processes using instances of the class *MessageQueue* (which was added by the CX package). The typical sequence of events for this animated simulation is shown in Example 1.

All of these interactions model quite closely (at least in my mind) the interactions between real people at a checkout counter. Note, however that much of the interaction I have included (that is, "take this item" and "got it!") is for the purpose of animating the simulation and serves no real purpose if the goal were merely to gather information about the simulated process. Figure 1 depicts a snapshot of this simulation while it is running.

Setting up

Setting up the sample simulation also makes use of some of the tools provided as part of the CX Multiprocessing

(continued from page 86)
Kit. In particular, the BitEditor is used to create a set of images used in the animation sequences. To do this, evaluate with *doit*, the Smalltalk expressions shown in Example 2.

Then, for each of the keys (*PersonUp*, etc.) evaluate the expression *BitEditor new openOn:(MarketImages at:'<key>')*. Create the animation images using those in Figures 1 – 4 as a guide. Because the BitEditor uses a fixed scale and can only handle bit maps up to about 60 pixels in height, the individual images should be magnified using the expression in Example 3. Finally, enter the simulation classes in the order shown in Figure 5.

Open the simulation window by evaluating the expression *MarketWindow new open*. The simulation is started by popping up the pane menu, START SIMULATION, and clicking on it.

Conclusions

The basic multiprocessing capabilities found in the Smalltalk/V 286 environment are sufficient, with some ingenuity, to create fairly elaborate simulations. The methods and classes included as part of the CX Multiprocessing Extension Kit offer many much needed enhancements to those limited capabilities. This is particularly true of the classes *EventQueue* and *Message Queue*, which facilitate the implementation of

```
Customer (upon removing an item from the cart):
            self
                        animate;  "Lean forward"
                        send:#takeItem to:checker.

Checker (upon receiving 'takeItem' command):
            self
                        animate;  "Turn to left";
                        send:#takeItem to:bagger;
                        send:#gotIt to:customer.

Bagger (upon receiving 'takeItem' command):
            self
                        animate;  "Turn to right"
                        send:#gotIt to:checker;
                        display.  "Turn back forward"

Customer (upon receiving 'gotIt' command):
            self
                        display.  "Stand up straight"

Checker (upon receiving 'gotIt' command from bagger)
            self
                        display;  "Turn back forward again"
                        send:#removeItemFromCart to:customer.

Customer (when the cart is empty):
            self
                        moveTo:counter exitPosition;  "Move to exit"
                        send:#nextCustomer to:counter.
```

Example 1: *Typical sequence of events in the checkout counter simulation*

```
MarketImages := Dictionary new:7.
MarketImages
            at:'PersonUp' put:(Form width:16 height:16).
MarketImages
            at:'PersonRight' put:(Form width:16 height:16).
MarketImages
            at:'PersonDown' put:(Form width:16 height:16).
MarketImages
            at:'PersonLeft' put:(Form width:16 height:16).
MarketImages
            at:'Customer' put:(Form width:16 height:32).
MarketImages
            at:'CustomerReaching' put:(Form width:16 height:32).
MarketImages
            at:'Counter' put:(Form width:32 height:64).
```

Example 2: *Smalltalk expressions to be evaluated with* doit

```
|oldImage newImage|
MarketImages keysDo:[:key|
            oldImage := MarketImages at:key.
    newImage := oldImage magnify:oldImage boundingBox
                                        by:2@2.
MarketImages at:key put:newImage].
```

Example 3: *Smalltalk expression used to magnify image*

timed events and interprocess communications. (When I first implemented the example application accompanying this article, I did so without using the message queuing capabilities. The result was extremely cumbersome and prone to unexplained failures. The decision to pass command messages between processes made the flow of information explicit and made the entire application much more understandable.)

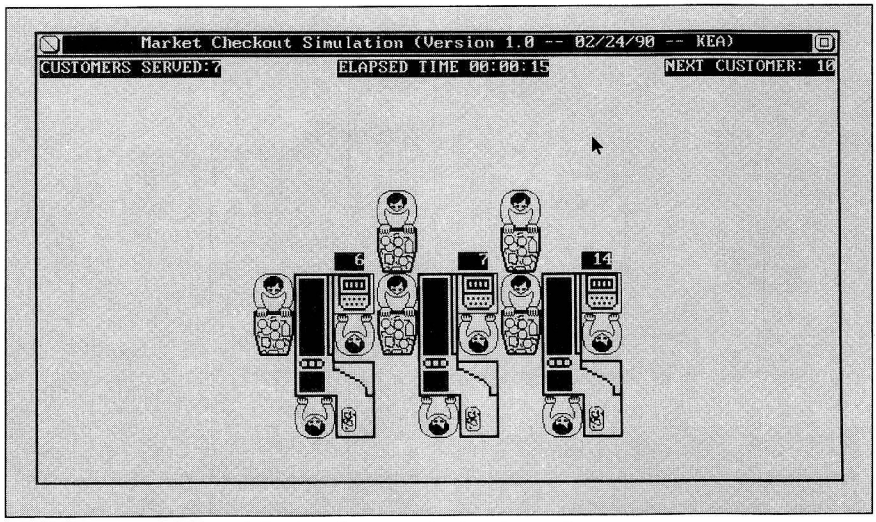


Figure 1: *Snapshot of the market simulation in action*

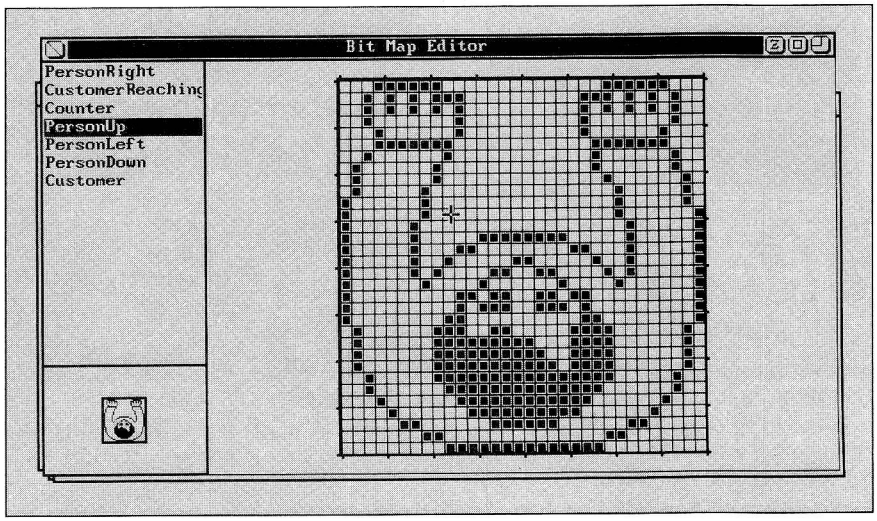


Figure 2: *Animation image for* PersonUp

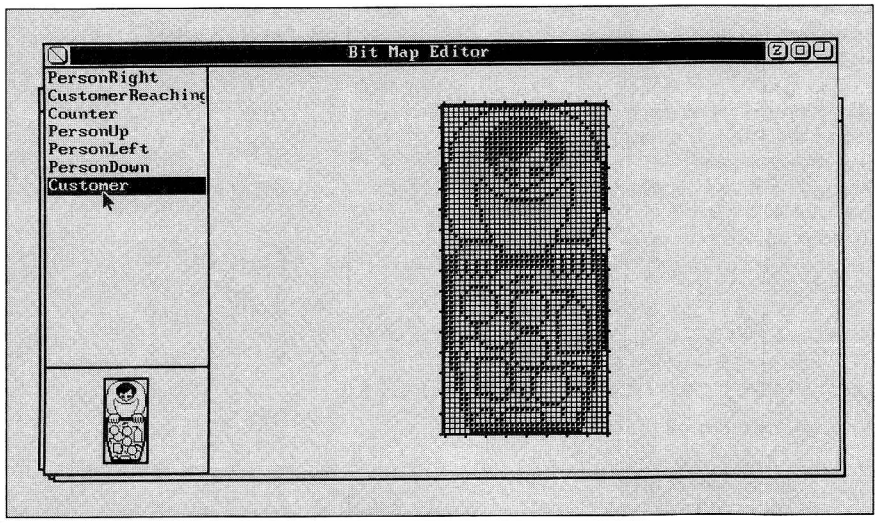


Figure 3: *Animation image for* Customer

The package offers an extraordinarily large number of features, all of which appear to be quite useful to the serious Smalltalk/V developer. Furthermore, there seems to be little evidence of redundancy or overkill. Some of the features, particularly the browser extensions, are worthy of a review by themselves.

One of the few real faults I could find was the lack of concrete examples. As seems to be common practice, many of the examples, while certainly complete and accurate, are rather trivial. To someone having limited experience with Smalltalk's multiprocessing features, that type of example fails to illustrate adequately the key concepts involved in creating and coordinating multiple processes. Personally, I would prefer to see one or two really complete examples than a hundred ways to print "hello world" using multiple processes.

From an operation point of view, the use of named processes and semaphores tends to impair Smalltalk's garbage collection and tends to leave inaccessible objects lying around unclaimed. In all fairness, this anomaly is documented; yet its occurrence is still disconcerting. I noticed that the use of the Process Status Window also seemed to produce a similar effect.

It is noted that the use of timed events creates a high-priority background process to watch the clock and look for events that need to be triggered. This appeared to affect the performance of other parts of my simulation, slowing it down considerably. Thus, for example, to implement a "sleep for n seconds" method, I found that watching the time myself produced better results than scheduling an event for n seconds in the future. Undoubtedly, for infrequent events occurring at some absolute date/time, this feature would be invaluable; but for short, repetitive delay operations it seemed to incur a considerable overhead.

Finally, though the cost of the extension package itself is certainly reasonable, the hidden costs of the Goodies, required to make full use of its capabilities, nearly triples the upfront cost of using this package. Even though the Goodies packages do offer a lot of added functionality at a reasonable price, I feel it only fair to warn the potential user that they represent an additional cost.

The net result of my experience exploring the CX Multiprocessing Extension Kit was certainly positive. I have gained additional insight into the realm of multiprocessing within the context

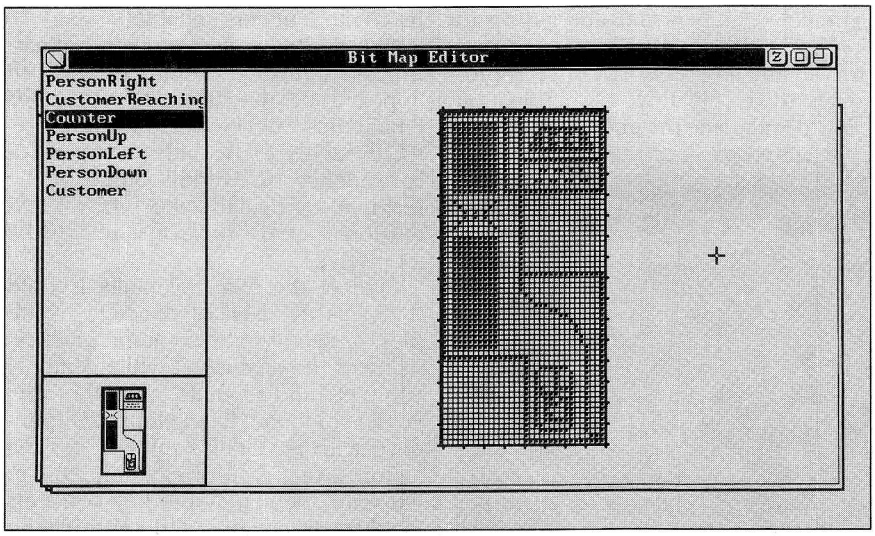


Figure 4: *Animation image for* Counter

RandomNumber
EmptyMenu
MarketActor
    Bagger
    Checker
    CheckoutCounter
    Customer
MarketSimulator
MarketWindow

Figure 5: *Hierarchy of classes for the market simulation*

Product Information
CX Multiprocessing Extension Kit
Computas Expert Systems
A.S (Veritasveien 1)
P.O. Box 410
N-1322 Hovik, Norway
Requirements: Smalltalk/V 2.0 or later
Goodies disks #1, #2, and #3
Smalltalk/V 286, 1.1 or later recommended
Price: $99.95

of Smalltalk and feel certain that potential users of this package would benefit from its diverse capabilities.

But for now, I'm going to get back to my simulated market and see if I can answer the question that has plagued modern man since the opening of the first supermarket: Why is the shortest line always the slowest?

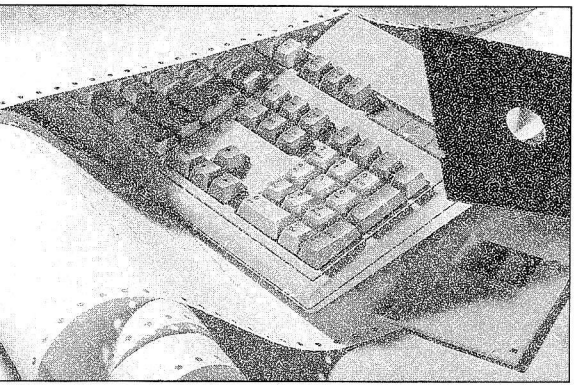
Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 114.)

Vote for your favorite feature/article.
Circle Reader Service **No. 8.**



Accessing Hardware from 80386 Protected Mode Part I

Understanding the 386 architecture may simply be a matter of building on what you already know

Stephen Fried

At one time, I ran a flight school that taught people how to fly aircraft and sailplanes. Of all the equipment we operated, the trickiest to manage was our fleet of Cessna L-19 "Bird Dogs," which we used to tow gliders and banners. The key to transitioning a pilot into an L-19 was getting the idea across that he wasn't flying an ordinary airplane, but one which had several distinct personalities. For example, if you limit the flaps to 30 degrees and the power to 150 HP, an L-19 flies just like a Skyhawk or C-170 (in fact, it has the same wing as both). However, when you go to full power, or full flap, what you get is an airplane that performs much like a helicopter. This split personality was designed into the L-19 for the Army, who used it for forward air control and covert operations in Vietnam. This same performance made it a great tow plane but a very expensive aircraft to transition pilots into. In fact, over a 15 year period we had three major accidents transitioning experienced pilots, and the Civilian Air Patrol (CAP) lost so many Bird Dogs that they were eventually forced to sell them off.

Without a shadow of a doubt, the "bird dog" of microprocessors has to be the 80386. In the case of the L-19, our experienced pilots could argue for hours about the right way to make a landing (where you put the flaps down, where you added power, and whether you three pointed it or landed on the mains). Transitioning the 386 from real to protected mode is every bit as complicated as making a short field landing in a Bird Dog. The process involves building tables in real memory, transitioning to protected mode, transitioning from 16- to 32-bit mode, building paging tables and, finally, transitioning to paged mode. The exact sequence used is a matter of personal choice, and at every stretch the processor and the assembly code that drives it has its own distinct personality. The resulting code is as difficult to decipher as any that has ever been written for a computer.

Stephen is the vice president of MicroWay's R&D. He is well known in the field for his PC numeric and HF chemical laser contributions. You can reach him at MicroWay Inc., P.O. Box 79, Kingston, MA 02364

Getting By

Fortunately, it is not necessary for ordinary folk to get involved in the writing of kernel routines. However, if you plan to directly access your 386 AT's facilities, such as the screen buffers, it will be necessary to understand the rudiments of 80386 memory management and how it affects application development.

Just how confused are people about the 80386? Our 80386 protected-mode compilers have been available for over two years, and our products have been used to port millions of lines of Fortran and C. Yet, as I discovered in writing this article, we had been making an incorrect claim that it was possible for the 80386 to run multiple segments, each of which could have up to 4 gigabytes of code or data. As we'll see shortly, this assertion is not really correct.

When I asked myself how I could write outstanding code for a processor that I didn't understand, I quickly came to the conclusion that it wasn't necessary to understand the 80386 to code it, but to just understand the two modes that virtually all 386 code runs in.

Protected Mode

The majority of 80386s running in PCs see two types of service: Real mode and 32-bit flat protected mode. 99.9 percent of all 386 applications, including those written with DOS extenders, Unix, Xenix, and OS/2, run in 32-bit flat protected mode. The mode is called flat or small because all of the code and data of a program exist in a single segment, which resembles the 32-bit address space of a typical mainframe. In the case of DOS extenders, the processors slip in and out of real mode to access MS-DOS.

Unix, Xenix, and the future 386 release of OS/2 run entirely in protected mode. Because these operating systems are either multitasking or multiuser, the protection of operating system facilities, and therefore all hardware, becomes a major issue. As a re-

sult, these operating systems make it impossible to write the kind of fast running "misbehaved" applications that are the subject of this article. They accomplish this by running the user's code at a low RPL (request privilege level) and making system facilities only accessible from code running at a high RPL. Therefore, the subject of this article applies to code running on DOS extenders only.

Probably the biggest problem with learning the 80386 is the fact that most of the books on the subject were written for or by operating system types. As it turns out, the 80386 has two sides: A complex one that takes months to fully appreciate and a simple, physical one that is an almost trivial extension of the 8086 architecture.

A 32-bit 8086

The easiest way to approach this multi-personality processor is to treat it like a 32-bit 8086 that can be attached to a piece of hardware that makes paged memory possible. To help facilitate this exposition, imagine that the year is 1984, that we work for Intel, and that we have been asked to design a 32-bit 8086. (We will ignore the fact that the 80286 exists and that we have been asked to create a processor that also runs most 80286 code.)

Recall that the 8086 has six general-purpose 8/16-bit registers which were adapted from similar 8- and 16-bit registers in the 8080. Also recall that the address space of the 8086 was extended to 20 bits by adding four 16-bit segment registers to the 8080 architecture that point to 64K-bytes windows, called "segments," that can be located on any of the 64K paragraphs (a modulo 16 address) that exist in a megabyte of physical memory.

There is a movement in the 386 extender industry towards running in ring 3 instead of ring 0

Segmentation was the trick that made the 16-bit registers of the 8086 capable of spanning a 20 bit address space. The problem we are now faced with is that we want to address more than a megabyte and we want to use 32-bit registers for computing addresses.

To generate an upward compatible architecture, we will now mimic what we did when we expanded the 8080 to 16-bits. We will use segments to act as windows into the address space and let our general-purpose registers contain offsets into the segment windows. We are now faced with several problems. Our segments must be capable of holding a lot of information, and to keep segments from hogging valuable address space, there should be some way to specify their size. Finally, to simplify upward compatibility, we will stay with 16-bit segment registers.

What comes out of these requirements is a segment that can be located at any system address in a 32-bit address space and whose size is not fixed, but specified by a 32-bit integer. Describing the size and location of a 32-bit segment in a 32-bit address space takes 64 bits of information and clearly violates our desire to leave our segment registers 16 bits wide. We resolve this by letting the segment register contain a 16-bit index into a table that is

stored in memory and contains the 8 bytes needed to describe our new 32-bit segment. This index is called a "selector," and the table it references a "descriptor table." The location of a segment will henceforth be called its "base," and we use the term "limit" to describe its size.

To make it possible for the processor to access descriptors quickly, we incorporate registers into the processor that hold the base and limit for all the currently active segments. We also expand the number of segment registers from the four of the 8086 (*cs*, *ds*, *ss*, *es*) to six, through two new data segments, *fs* and *gs*.

Attribute Bits

To implement protection, we must free up a few of the 64 bits that we dedicated to the descriptors above. We do this by reducing the size of the limit from 32 to 20 bits. This limits the size of a segment to a megabyte, so we use one of the 12 bits that have just been freed up to specify the granularity of a segment. When this bit is set to zero, the segment is said to have "byte granularity" and its limit is a 20-bit integer. When this bit is 1, the limit value gets multiplied by 4K-bytes, yielding a 4-gigabyte upper value to the limit, with 4K granularity.

The other 11 bits that we have carved from our original 32-bit limit are used to specify protection attributes. These include bits that describe whether the segment is a 16- or 32-bit attribute (that is, the processor has a 16- and 32-bit default mode specified by this bit), the privilege level of the segment (0..3), a "present" bit (if this bit is not set, the selector is invalid), a DT bit that distinguishes ordinary memory segments from those that describe system resources (which are ignored here), and four TYPE bits that specify 16 possible usages for a segment (read-only, read/write, execute-only, execute/read).

Of the 12 attribute bits, the only ones that we will encounter are those that specify the granularity, protection level, and segment TYPE. The attribute bits are used by the processor to ensure that every time a byte is accessed from memory, the program accessing the byte has the right to access the byte in question, that the byte lies in the segment, and that it is being used for its intended purpose (you can only execute code, not data, and vice versa).

As a result of the depth of protection provided by the processor, bugs which would cause crashes in 8086 systems, cause, more often than not, memory protection faults in 80386 systems. This makes debugging much easier. You simply run the program under an 80386 debugger and when the processor hits the fault, examine the program to determine what caused an illegal access request. It is usually impossible to backtrack after an 8086 error because the original error destroys the processor stack, which causes the CPU to jump to data instead of code, resulting in everything becoming scrambled. Errors such as stack underflows cause immediate exceptions in the 80386, making it possible to backtrack before the processor has destroyed the information needed to track down the bug.

As a result of the depth of protection provided by the processor, bugs that would cause crashes in 8086 systems cause memory protection faults in 80386 systems

General Registers

So far we have spent all of our time worrying about how to extend the concept of a 64K segment into a general-purpose 32-bit segment. Now that we have created a 32-bit segmented framework for accessing information in memory, we must worry about extending the size of the six 8086 general-purpose registers: *ax*, *bx*, *cx*, *dx*, *si*, and *di*. We will do the same thing with them that we did when we extended the 8080 architecture from 8 to 16 bits. We create a new 32-bit register for each, having its lowest 16 bits named after the corresponding 8086 register, and its two lowest bytes named after the 8-bit registers of the 8086. For example, the 16-bit register *ax*, which contains the 8-bit registers *al* and *ah* in the 8086, gets expanded in the 80386 into a 32-bit register *eax*, which has a 16-bit component *ax* and two 8-bit components *al* and *ah*.

The registers *ebx*, *esi*, and *edi* are used in exactly the same manner as *bx*, *si*, and *di*. We also add some new addressing modes that simplify accesses of vectors but, otherwise, our architecture looks remarkably like an 8086. Addresses are computed in these 32-bit registers and are used as offsets into the 1- to 4-gigabyte segments that we developed earlier. Intersegment jumps and calls are NEAR in the 80386, but when running in 32-bit mode, NEAR changes its meaning from the 16 bits of the 8086 to 32 bits.

Supporting Syntax

To simplify the encoding of instructions, we use the same opcodes for *mov eax, eax* as we did for *mov ax, ax*, and so on. The size of the register operands is determined in two ways. One of the attribute bits in the segment descriptor describes a segment as being 16 or 32 bits. In addition, when the processor is running in 16-bit mode, a prefix byte can be used ahead of an instruction to indicate that the operands of that instruction are only 32 bits. A similar prefix makes it possible to use 16-bit registers in 32-bit mode.

The use of an override prefix makes it possible to write 16-bit code, which accesses the 32-bit registers. However, when running in real mode, accessing 32-bit registers does not buy much, as the size of segments in real mode is limited to 64K and the address space is limited to the first megabyte. In fact, we create real mode to make it possible to run 8086 code without doing anything, and to provide an execution environment for setting up descriptor tables in memory, so that the processor is capable of setting itself up before jumping into protected mode.

Other Features

There are a few other features that I should at least mention. The overlooked details include several control registers, three types of descriptor tables, task segment switches (48-bit intrasegment FAR calls), paging tables, and 8086 virtual mode. One of the facilities that we have to mention, the IDT (interrupt descriptor table) makes it possible for the processor to create different interrupt tables for different tasks.

These rather abstract facilities make it possible for these two personality processors to use software control to exhibit many other personalities (most of which will never see the light of day in the real world). In addition, they make it possible to implement demand-paged virtual memory that is very efficient. Virtual memory is available for all of the operating systems and environments that Microway's NDP C works with, making it possible to run mainframe programs on 386 systems that only have 1 to 2 Mbytes of RAM and a lot of free space on a hard disk.

A Quick look at the Map

The 80386 has a "physical" side that is quite close to the physical side presented by the 8086, and an abstract side that we can ignore. We will now examine this physical side and make the connection between the environment of our protected-mode application and the real-mode resources that the processor takes advantage of for doing I/O in an 80386 "AT" system.

Figure 1 shows the local descriptors for a program running under Phar Lap with the no page switch on. These values were obtained by running an NDP C program under the Phar Lap 386DEBUG program and using the *dl* command to dump the local descriptor table. The selector numbers on the left side of the table are the values that a programmer passes into the 386 segment registers to activate a segment. Because 386DEBUG was invoked with paging off, the BASE values in Figure 1 correspond to physical addresses.

The memory map has a number of these selectors pinpointed on its left side. Looking at selectors 0C and 14, we see that their corresponding segments are located at the start of what IBM calls "extended memory" (the start of the second megabyte of memory). If we had invoked 386DEBUG with paging on, the primary difference in our segment memory map would be that 0C and 14 would be moved down into the first megabyte to save memory. However, with paging enabled, it would not be possible to read the physical location of a segment from the selector BASE value, as the processor performs an additional address translation with paging enabled. Therefore, we will examine some of the selectors in Figure 1 that have been set up by the DOS extender before going on to see what happens when paging is enabled.

Selector 1C has been set up so that it contains the current screen buffer. This selector has a base that starts at address 0B800:0 (in 8086 notation) and is 0FFFH + 1 byte in length (16K bytes). The fact that this segment corresponds exactly to the screen buffer was no accident. The Phar Lap DOS Extender queried the system to find out what kind of graphics adapter was active, and based on this information created an entry in the LDT (local descriptor table) that precisely matched the device. It is also important to point out that the use of selector 1CH is preferred over selector 34H (which maps in the entire first megabyte of RAM) for screen buffer accesses, because an out-of-bounds write will result in a protection fault when using 1CH, but could have disastrous results if 34H were used.

The selectors 0CH and 14H were created for user code and data. Note that these selectors have the same location base and limit. In fact, they are identical in every way, except for the attribute flags. The format of the attribute byte is:

```
    upper                lower
¦P¦DPL¦DT¦              ¦TYPE¦
```

Looking over the memory map, we see that the flag byte has only two values: 92H and 9AH. The lower nibble in 92H indicates that the segment is of type 2, which means the segment is read/write (for data only). All but one of the segments must therefore contain data. Looking at the map, we discover that the segment that we have identified with "user code" has an attribute of 9A. The TYPE nibble, 0AH, indicates that selector, 0CH, is execute/read only (code).

The upper nibble contains miscellaneous information about the segment, including the present bit, two bits that specify the privilege level, and a bit which, when set, specifies that the descriptor describes memory (as opposed to a task switch or special system entity). The binary translation of 9 is 1001, which translates into the segment marked as present in memory with a privilege level of 0. Privilege level 0 is the highest available, and is frequently referred to as "ring 0."

Segments that run in ring 0 are theoretically capable of creating havoc by playing games with systems' tables that should only be accessed by the operating system or DOS extender. As a practical matter, the only time we have had to deal with invisible system tables, such as the global descriptors, was in the early 80386 days, before the DOS extenders had calls for mapping in new hardware, such as the Weitek coprocessor (which is now automatically mapped in by all DOS extenders).

As long as the program you write goes through systems calls provided by Phar Lap and Eclipse to modify lower-level system tables, such as the interrupt descriptor table, the program that results will conform to the VCPI specification, which means it will run with VCPI operating environments, such as Desqview-386, Netware-386, Phar Lap, and Eclipse.

As a point of interest, Eclipse runs programs in ring 3. There is a movement in the 386 extender industry toward running in ring 3 instead of ring 0. As long as the operating environments continue to provide the memory mapping capabilities that are utilized below, we have no objection to running in ring 3 over 0. However, we think there is, and will continue to be, a need for operating environments that provide direct access to all system resources, as a counter measure to operating systems such as OS/2 and Unix, which are attempting to shut off access to these facilities.

Real Memory from Protected Mode

To move a block of characters and attributes into screen RAM in an 8086 system, we might employ a block move. This technique is frequently used by spreadsheets that build an image in memory of what the screen is going to contain and then instantaneously move this buffer to screen RAM by using a single processor instruction. To set up a block move in an 8086, we point the *ds:si* registers at the source, the *es:di* registers at the destination, place the number of bytes to be moved in *cx* and then use a *rep movsb* instruction to have the processor make the transfer for us.

The code for an 80386 block move is identical, except that we now use 32-bit registers to hold 32-bit offsets, and where we used physical paragraphs in *ds* and *es*, we now use the appropriate selectors. In addition, where we placed the count in *cx*, we now place the count in *ecx*, which is a 32-bit register and makes it possible to move more than 64K with a single instruction. For example, to move a 16K buffer of character attribute pairs to a monochrome screen buffer located at paragraph B800, we would employ one of the two sequences of code shown in Figure 2, depending on whether we were running in real mode or 80386 32-bit mode under Phar Lap.

The program assumes that the buffer being moved is contained by the current data segment in *ds*. It then sets up a FAR pointer to the destination (screen buffer at B800:0). Note that where the real-mode code used the physical paragraph of the screen buffer, the 80386 uses the selector set up by Phar Lap. Next, the code points *si* or *esi* at the buffer to be moved. Again, note that where a 16-bit offset was used by the real mode code, a 32-bit offset is now being used by the 80386 for the 32-bit code. Finally, the program sets the number of bytes to be moved in *cx* or *ecx*, and requests the processor to carry out the block move. Except for the first line, these two sequences are virtually identical.

Because the selectors in Figure 1 can access all of the memory in the first physical megabyte of RAM, we have just demonstrated that it is possible to access all of a system's "real" memory from a program running in protected mode. In our example, the source buffer is contained by the default data segment, 14H, which is located in "extended" memory above the first megabyte.

Selector	BASE	Limit	Flags	Use	Gran	Comment
04	53030	FF	92	32	BYTE	DOS EXTEND
0C	100000	2FF	9A	32	BYTE	USER CODE
14	100000	2FF	92	32	BYTE	USER DATA
1C	B8000	FFF	92	32	BYTE	DOS SCREEN
24	53030	FF	92	32	BYTE	DOS EXTEND
2C	52f60	B9	92	32	BYTE	DOS EXTEND
34	00000000	FFFFF	92	32	BYTE	1st MEG
3C	C0000000	FFFF	92	32	BYTE	WEITEK

Figure 1: *The local descriptors for a program running under Phar Lap*

Real mode		32-bit protected mode		
mov	ax,0B800H	mov	eax,1CH	; set destination
mov	es,ax	mov	es,ax	; segment
xor	di,di	xor	edi,edi	; dest offset = 0
mov	si,buffer	mov	esi,buffer	; set source offset
mov	cx,1000H	mov	ecx,1000H	; set count
rep	movsb	rep	movsb	; perform block move

Figure 2: *16-bit vs. 32-bit assembly code to move a 16K buffer of character attribute pairs to a monochrome screen buffer*

(Continued from page 96)

48-bit Address Space?

All that remains to our expose of the 386's flat model is to explore the operation of ports, interrupts, and paging. However, before we leave segmentation, there is one myth we need to burst. The typical text on the 80386 presents the processor as having three address spaces — virtual, linear, and physical. Up to this point, what we have been exploring is the linear and physical, which are both identical when paging is disabled. The mythical address space turns out to be the "virtual" one. The myth was born because individuals who were used to programming in the large or huge models on the 8086 asked, "What would happen if we could write large or huge code on an 80386, instead of small code?" They quickly came to the conclusion that programs written with compilers, and operating systems that support 48-bit pointers (the 16-bits of the selector count for 16- and the 32-bit maximum size of the limit count for 32), would be capable of addressing a 48-bit address space, which just happens to contain 64 terrabytes!

To move a block of characters and attributes into screen RAM in an 8086 system, we might employ a block move

We don't know who created this concept, although we suspect that Intel marketing told its systems' architects (after the last perceived black eye they got from a segmented architecture) that if they had to resort to segmentation again, they better have a damn good reason. The reality of the situation is that practical program size is limited by the size of what Intel calls the "linear" address space (to 32-bits), and that a 48-bit address space will not become a reality until Intel increases the size of the linear address space in a future device.

To prove the point, we did a calculation of what would happen if we took a simple program that performed a matrix multiply and extended it to handle arrays whose total size was greater than 4 gigabytes. As the total size of the arrays in our problem approach 4 gigabytes (each of the three arrays approach 1.3 gigabytes), we have to abandon our 80386 small model, and Phar Lap, in favor of a compiler-supported memory model and operating system that utilizes the virtual address space (which is not the same as demand-paged virtual memory, which we commonly refer to as "virtual memory").

Once our problem hits the 1.4-gigabyte array size, it is impossible to have all three arrays in our 4-gigabyte linear address space at the same time. So, we take advantage of the present bit in the descriptor table to make it possible for our large model operating system to swap arrays as needed. Our large model operating system makes it possible to run large model virtual segments. When we compute the time required to swap our 1.4-gigabyte segments as required by our algorithm, we discover that, assuming we have the world's fastest hard disks, our code runs 100,000 times slower than it did in the small model currently supported by Phar Lap, Unix, and Xenix.

The largest sized array that our large model supports is 4 gigabytes, which means our problem will span a tiny (in comparison to 64 terrabytes) 12-gigabyte address space. But never fear, we have still not finished digging into our bag of 8086 tricks. By resurrecting FAR pointers, the huge model, and tiling, we can hit our 64 terrabyte goal — and for only a cost factor of 400 percent in code efficiency.

What's Next?

That these systems tricks are crucial for future Intel products is quite evident from the 80486, which, unlike the 80386, achieves its best speed with small model code that limits data accesses to the *ds* segment register only. It's amazing what happens to the best laid plans of product managers, public relations, and system types, when everyone suddenly discovers that the key to selling systems is simplicity (i.e., RISC)! But, I hope to convince you next month in Part II of this article that the only use for FAR pointers in 80386 code appear in operating system kernels.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 7.**

Listing One *(Text begins on page 30.)*

```
// GETDATA.C    Read GBLDATA.OBJ to get global variables for DLLs
//      02/12/89        by Gary Syck
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
#include <stdlib.h>
#include <string.h>
#include <dos.h>
#include "dll.h"

// Open GBLDATA and read data
void
GetData()
{
        int fd;                         // File descriptor for GBLDATA.OBJ
        int     Idx;                    // Index numbers from the file
        unsigned long Elmnts, DSize;    // Number and size of global data items
        unsigned PubGrp, PubSeg;        // Group and segment indexes for data
        unsigned DOff;                  // Offset for initialized data
        unsigned char type;             // type of record read
        int     size;                   // size of the record
        unsigned char *Data;            // the data to read
        unsigned char *DSptr;           // Pointer to the Data segment
        int i, j;

        if((fd=open( "gbldata.obj", O_BINARY|O_RDONLY )) == -1 )
        {
                printf( "Unable to open file\n" );
                exit(1);
        }
        DataSize = 0;
        SymCnt = 0;
        AllocNumb = 0;
        SymSize = 0;
        type = 0;
        DSptr = &DataSpace;
        while( type != MODEND )
        {
                read( fd, (char *) &type, sizeof( unsigned char ));
                read( fd, (char *) &size, sizeof( int ));
                Data = malloc( size );
                read( fd, Data, size );
                switch( type )
                {
                        case PUBDEF:    // The record contains public symbols
                                i=0;
                                if( Data[i]&0x80 )
                                {
                                        PubGrp = (Data[i++]&0x7f)<<8;
                                        PubGrp += Data[i++];
                                }
                                else
                                        PubGrp = Data[i++];
                                if( Data[i]&0x80 )
                                {
                                        PubSeg = (Data[i++]&0x7f)<<8;
                                        PubSeg += Data[i++];
                                }
                                else
                                        PubSeg = Data[i++];
                                if( PubSeg == 0 )
                                        i += 2;  // skip the frame number
                                AllocateSyms(); // make memory for all symbols
                                /* assume all public defs are in the DGROUP */
                                while( i<size-1 )
                                {
                                  Syms[SymCnt].Name = malloc( Data[i] + 2 );
                                  strncpy( Syms[SymCnt].Name, &Data[i+1],
                                                                  Data[i] );
                                  Syms[SymCnt].Name[Data[i]] = '\0';
                                  i += Data[i]+1;
                                  Syms[SymCnt].Seg = FP_SEG(DSptr);
                                  Syms[SymCnt].Offset = (FP_OFF(DSptr))+
                                                        *((int *) &Data[i] );
                                  SymCnt++;
                                  i += 2;
                                  if( Data[i]&0x80 ) // skip over the type
                                  {
                                        Idx = (Data[i++]&0x7f)<<8;
                                        Idx += Data[i++];
                                  }
                                  else
                                        Idx = Data[i++];
                                }
                                break;
                        case LEDATA: // record contains data for data segment
                                i = 0;
                                if( Data[i]&0x80 )
                                {
                                        Idx = (Data[i++]&0x7f)<<8;
                                        Idx += Data[i++];
                                }
                                else
                                        Idx = Data[i++];
                                /* Assume all data is for the data segment */
                                DOff = *((int *) &Data[i] );
                                i += 2;
                                memcpy( &DSptr[DOff], &Data[i], size-(i+1) );
                                if( DataSize < DOff + size - (i+1))
                                        DataSize = DOff + size - (i+1);
                                break;
                        case COMDEF:  // record contains uninitialized data
                                i = 0;
                                while( i < size-1 )
                                {
                                 Syms[SymCnt].Name = malloc( Data[i]+2 );
                                 strncpy( Syms[SymCnt].Name, &Data[i+1],
                                                                  Data[i] );
                                 Syms[SymCnt].Name[Data[i]] = '\0';
                                 i += Data[i] + 2;
                                 if( Data[i++] == 0x61 )
                                {
                                        if( Data[i] < 128 )
                                          Elmnts = (unsigned long) Data[i++];
                                          else
                                          {
                                          j = Data[i++] - 127;
                                          Elmnts = 0L;
                                          memcpy( &Elmnts, &Data[i], j );
                                          i += j;
                                          }
                                         if( Data[i] < 128 )
                                         DSize = (unsigned long) Data[i++];
                                         else
                                         {
                                         j = Data[i++] - 127;
                                         DSize = 0L;
                                         memcpy( &DSize, &Data[i], j );
                                         i += j;
                                         }
                                         Syms[SymCnt].Size = (unsigned)
                                                                (Elmnts * DSize);
                                         if( (unsigned long) SymSize +
                                              (unsigned long) (Elmnts *
                                                               DSize) >= 32000L )
                                         AllocateSyms();
                                         SymSize += (unsigned) (Elmnts *
                                                                      DSize);
                                         SymCnt++;
                                        }
                                }
                                break;
                        default:
                                break;
                }
                free( Data );
        }
        close( fd );
        AllocateSyms(); // make memory for all of the symbols
}

// make a memory block to hold the previous symbols
void
AllocateSyms()
{
        char *Buff;
        unsigned Seg, Off;
        if( SymSize )
        {
                Buff = malloc( SymSize );
                Seg = FP_SEG( Buff );
                Off = FP_OFF( Buff );
                SymSize = 0;
                while( AllocNumb < SymCnt )
                {
                        Syms[AllocNumb].Seg = Seg;
                        Syms[AllocNumb].Offset = Off;
                        Off += Syms[AllocNumb].Size;
                        AllocNumb++;
                }
        }
}
```

End Listing One

Listing Two

```
; STUB.ASM  Used by DLL to see if a function needs to be loaded.
; 02/12/89  By Gary Syck

        TITLE   stub.asm
        NAME    stub

        .8087
STUB_TEXT       SEGMENT  WORD PUBLIC 'CODE'
STUB_TEXT       ENDS
_DATA   SEGMENT  WORD PUBLIC 'DATA'
_DATA   ENDS
CONST   SEGMENT  WORD PUBLIC 'CONST'
CONST   ENDS
_BSS    SEGMENT  WORD PUBLIC 'BSS'
_BSS    ENDS
DGROUP  GROUP   CONST, _BSS, _DATA
        ASSUME  CS: STUB_TEXT, DS: DGROUP, SS: DGROUP
EXTRN   _LoadFunc:FAR           ; Function to get a function into memory
EXTRN   _FuncLst:DWORD          ; The function table
_DATA   SEGMENT
        PUBLIC  _DataSpace
_DataSpace      db 32000 dup(?)
_DATA   ENDS

STUB_TEXT       SEGMENT
        ASSUME  CS: STUB_TEXT
        PUBLIC  _Stub
_Stub   PROC FAR
        mov     bx,offset _FuncLst      ; this is modified before use
        mov     ax,seg _FuncLst
        mov     es,ax
        mov     ax,WORD PTR es:[bx+4]   ; check .Loc
        or      ax,WORD PTR es:[bx+6]
        jne     noload
        push    es                      ; save ES and BX
        push    bx
        call    FAR PTR _LoadFunc
```

```
        pop     bx
        pop     es
noload:
        jmp     DWORD PTR es:[bx+4]     ; go to the function
_Stub   ENDP
STUB_TEXT       ENDS
END
```

End Listing Two

Listing Three

```
// LINKER.C Link a module at run time
// 02/12/89 by Gary Syck
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <io.h>
#include <stdlib.h>
#include "dll.h"

// Load the function in the FUNCTAB entry
void
LoadFunc( FUNCTAB *func )
{
        char FileName[15], *p, Name[80];
        int     fd;
        unsigned char type;
        int size;
        unsigned char *Data, *Dp, *LastFunc;
        int i, j, ln;
        unsigned pubgrp, pubseg, dataseg;
        unsigned DROff, Loc, FixDat;
        unsigned FrameIdx, TargetIdx, TargetDisp, tmp;
        unsigned long ltmp;
        int dataoff;
        unsigned LocalSize;     // size of the data allocated here
        struct {
                int     Flag;           // 1 = data, 2 = function
                int SymNumb;    // What symbol
        } ExtSyms[20];
        int ExtCnt;
        struct {
                int FuncNumb;
                int OffSet;
        } Pubs[10];
        int PubCnt;
        struct {
                unsigned Meth;  // The method to use
                unsigned Idx;   // The index
        } Threads[8];
        void *ptr;
        unsigned char *DSptr;
        unsigned constseg;      // Where constants begin

        strncpy( FileName, &func->Name[1], 14 );
        FileName[14] = '\0';
        if((p=strchr( FileName, '#' )) != NULL )
                *p = '\0';
        strcat( FileName, ".obj" );
        if((fd=open( FileName, O_RDONLY|O_BINARY )) == -1 )
        {
                printf( "Unable to open file: %s\n", FileName );
                return;
        }
        LocalSize = 0;
        ExtCnt = 0;
        PubCnt = 0;
        type = 0;
        DSptr = &DataSpace;
        while( type != MODEND )
        {
                read( fd, (char *) &type, sizeof( unsigned char ));
                read( fd, (char *) &size, sizeof( int ));
                Data = malloc( size );
                read( fd, Data, size );
                switch( type )
                {
                        case THEADR:    // ignore the header
                        case COMMENT:   // ignore comments
                        case GRPDEF:    // these are always the same
                        case MODEND:
                        case SEGDEF:
                        case LNAMES:
                                break;
                        case EXTDEF:    // find external names
                                for( i=0; i<size-1; i += ln+2 )
                                {
                                  ln = Data[i];
                                  if( ln )
                                  {
                                        strncpy( Name, &Data[i+1], ln );
                                        Name[ln] = '\0';
                                        for( j=0; j<SymCnt && strcmp(
                                                        Syms[j].Name, Name );j++);
                                        if( j<SymCnt )
                                        {
                                        ExtSyms[ExtCnt].Flag = 1;
                                        ExtSyms[ExtCnt].SymNumb = j;
                                        }
                                        else
                                        {
                                        for( j=0; j<FuncCnt
                                        && strcmp( FuncLst[j].Name,Name );j++);
                                        if( j>=FuncCnt ) // make space for it
                                            {
                                            FuncLst[FuncCnt].Name = malloc
                                                        (strlen(Name)+2 );
                                             strcpy( FuncLst[FuncCnt].Name, Name );
                                            FuncLst[FuncCnt].Loc = NULL;
                                            FuncLst[FuncCnt].Flag = 0;
                                            memcpy( FuncLst[FuncCnt].Stub,
                                                        Stub, STUBSIZE );
                                            memcpy(&FuncLst[FuncCnt].Stub[4]
                                                        ,&FuncLst[FuncCnt],2);
                                                        FuncCnt++;
                                                }
                                                ExtSyms[ExtCnt].Flag = 2;
                                                ExtSyms[ExtCnt].SymNumb = j;
                                        }
                                        ExtCnt++;
                                        }
                                }
                                break;
                case PUBDEF:    // add to list of available functions
                                i = 0;
                                if( Data[i]&0x80 )
                                {
                                        pubgrp = (Data[i++]&0x7f) << 8;
                                        pubgrp += Data[i++];
                                }
                                else
                                        pubgrp = Data[i++];
                                if( Data[i]&0x80 )
                                {
                                        pubseg = (Data[i++]&0x7f) << 8;
                                        pubseg += Data[i++];
                                }
                                else
                                        pubseg = Data[i++];
                                if( pubseg == 0 )       // skip the frame
                                        i += 2;
                                while( i < size-1 )
                                {
                                        ln = Data[i];
                                        if( ln )
                                        {
                                             strncpy( Name, &Data[i+1], ln );
                                             Name[ln] = '\0';
                                        }
                                        i += ln + 1;
                                        memcpy( &ln, &Data[i], sizeof( int ));
                                        i += 2;
                                        if( Data[i]&0x80 )
                                                i += 2;
                                        else
                                                i++;
                                        for( j=0; j<FuncCnt
                                             && strcmp( FuncLst[j].Name,
```

(Listing continued on page 106)

Listing Three *(Listing continued, text begins on page 30.)*

```
                                                Name ); j++ );
                        Pubs[PubCnt].FuncNumb = j;
                        Pubs[PubCnt].OffSet = ln;
                        PubCnt++;
                }
                break;
        case LEDATA:
                i = 0;
                if( Data[i]&0x80 )
                {
                        dataseg = (Data[i++]&0x7f) << 8;
                        dataseg += Data[i++];
                }
                else
                        dataseg = Data[i++];
                memcpy( &dataoff, &Data[i], sizeof( int ));
                i += sizeof( int );
                if( dataseg == 2 ) // it's for the data segment
                {
                        Dp = &DSptr[DataSize+dataoff];
                        memcpy( Dp, &Data[i], size - ( i+1 ));
                        if( LocalSize < dataoff + size - (i+1))
                            LocalSize = dataoff + size - (i+1);
                        constseg = LocalSize+DataSize;
                }
                else if( dataseg == 3 ) // for const segment
                {
                        Dp = &DSptr[constseg+dataoff];
                        memcpy( Dp, &Data[i], size - ( i+1 ));
                        if( LocalSize < dataoff + size - (i+1))
                            LocalSize = dataoff + size - (i+1);
                }
                else    // here is the code
                {
                        if( dataoff == 0 )
                             LastFunc = malloc( size - (i+1) );
                        else
                          LastFunc = realloc( LastFunc, size
                                        - (i+1) + dataoff );
                        Dp = &LastFunc[dataoff];
                        memcpy( Dp, &Data[i], size-(i+1));
                        for( j=0; j<PubCnt; j++ )
                        {
                                FuncLst[Pubs[j].FuncNumb].Loc
                                 = &Last- Func[Pubs[j].OffSet];
                        }
                }
                break;
        case FIXUPP: // only look at the fixup fields all
                     // threads are the same.
                i = 0;
                while( i < size-1 )
                {
                        if( (Data[i])&0x80 )    // its a fixup
                        {
                         DROff = ((Data[i]&3)<<8) + Data[i+1];
                         Loc = Data[i];
                         i += 2;
                         FixDat = Data[i++];
                         FrameIdx = TargetIdx = TargetDisp = 0;
                         if( !(FixDat&0x80) && (FixDat&0x70)
                                                != 0x50 )
                                /* there is a frame index */
                         {
                                if( Data[i]&0x80 )
                                {
                               FrameIdx = (Data[i++]&0x7f) << 8;
                               FrameIdx += Data[i++];
                         }
                                else
                                        FrameIdx = Data[i++];
                         }
                         if( !(FixDat&8) ) /* thread index */
                         {
                         if( Data[i]&0x80 )
                         {
                         TargetIdx = (Data[i++]&0x7f) << 8;
                                  TargetIdx += Data[i++];
                         }
                         else
                         TargetIdx = Data[i++];
                         }
                         if( !(FixDat&4) )
                         {
                         memcpy( &TargetDisp, &Data[i],
                                                sizeof( int ));
                             i += 2;
                         }
                         /* fix up FixDat from threads */
                         if( FixDat&0x80 ) // frame from thread
                         {
                             j = ((FixDat&0x70)>>4) + 4;
                             tmp = Threads[j].Meth << 4;
                             FixDat = (FixDat&0xf) | tmp;
                                FrameIdx = Threads[j].Idx;
                         }
                         if( FixDat&8 ) // target from a thread
                         {
                            j = FixDat&3;
                            tmp = Threads[j].Meth&3;
                            FixDat = (FixDat&0xf4) | tmp;
                            TargetIdx = Threads[j].Idx;
                         }
                         switch( Loc&0x1C ) // find what we need
                         {
                         case 0x4:       // offset fixup
                            if( (FixDat&7) == 4 )
                                 {
                            /* get the value to be fixed */
                                 memcpy( &tmp, &Dp[DROff],
                                                sizeof(int));
                               if( TargetIdx == 2 ) // data seg
                                        {
                                     tmp += ((unsigned long)
                                        &DSptr[DataSize])&0xffff;
                                        }
                                 else if( TargetIdx == 3 )
                                        {
                                    tmp += ((unsigned long)
                                      &DSptr[constseg])&0xffff;
                                        }
                               /* put the fixed number back */
                               memcpy( &Dp[DROff], &tmp,
                                                sizeof(int));
                                        }
                                 else if((FixDat&7) == 6 )
                                        {
                                 if( !(Loc&0x40) )
                                   {
                                   ltmp = (unsigned long)
                                   (FuncLst[ExtSyms[TargetIdx-1]
                                    .SymNumb].Loc);
                                    ltmp -= (unsigned long)
                                                (&Dp[DROff])+2L;
                                         memcpy( &Dp[DROff],
                                            &ltmp, sizeof(int));
                                   }
                                    else // put the offset in
                                   {
                                   memcpy( &Dp[DROff],
                                    &Syms[ExtSyms[TargetIdx-1]
                                    .SymNumb].Offset, sizeof(
                                                        int ));
                                   }
                                 }
                                 break;
                                 case 0x8:  // segment fixup
                                 if( (FixDat&7) == 6 )
                                 {
                                 memcpy( &Dp[DROff]
                                  , &Syms[ExtSyms[TargetIdx-1]
                                  .SymNumb].Seg, sizeof( int ));
                                 }
                                 break;
                                 case 0xC: // symbol from target
                                 if( (FixDat&7) == 6 )
                                 {
                                 ptr = FuncLst[ExtSyms
                                    [TargetIdx-1].SymNumb].Stub;
                                    memcpy( &Dp[DROff], &ptr,
                                                sizeof( void *));
```

(Listing continued on page 108)

Listing Three *(Listing continued, text begins on page 30.)*

```
                                        }
                                        break;
                                        }
                                }
                                else    // its a thread
                                {
                                        j = Data[i]&3;
                                        if( Data[i]&0x40 )
                                                j += 4;
                                   Threads[j].Meth = (Data[i]&0x1C)>>2;
                                   i++;
                                   if( Data[i]&0x80 )
                                   {
                                        tmp = (Data[i++]&0x7f) << 8;
                                        tmp += Data[i++];
                                   }
                                    else
                                        tmp = Data[i++];
                                    Threads[j].Idx = tmp;
                                }
                        }
                        break;
                default:
                 printf( "invalid record: %x size: %d\n", type, size );
                 break;
                }
                free( Data );
        }
        close( fd );
        DataSize += LocalSize;
}
```

End Listing Three

Listing Four

```
// DLL.C Implement DLLs for DOS
// 02/12/89 by Gary Syck
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAIN
#include "dll.h"

void
main( int argc, char *argv[] )
{
        void (*func)(void);
        unsigned char *ptr;

        GetData();                              // Read the global data
        FuncCnt = 2;                            // make two functions
        FuncLst[0].Name = malloc( 10 );
        strcpy( FuncLst[0].Name, "_printf" );   // The library function printf
        FuncLst[0].Loc = printf;
        FuncLst[0].Flag = 1;
        memcpy( FuncLst[0].Stub, &Stub, STUBSIZE );
        ptr = FuncLst;
        memcpy( &FuncLst[0].Stub[1], &ptr, 2 );

        FuncLst[1].Name = malloc( 10 );
        strcpy( FuncLst[1].Name, "_main" );       // The first function to run
        FuncLst[1].Loc = NULL;
        FuncLst[1].Flag = 0;
        memcpy( FuncLst[1].Stub, &Stub, STUBSIZE );
        ptr = &FuncLst[1];
        memcpy( &FuncLst[1].Stub[1], &ptr, 2 );
        func = FuncLst[1].Stub;
        (*func)();      // Call the main function
}
```

End Listing Four

Listing Five

```
// Global data to be used with sample DLL
// make various data items

char Str[] = "Tester equals";
int Flubber=20;
int Tester=14;
```

End Listing Five

Listing Six

```
// sample DLL routine
#include <stdio.h>

void main(void);
void Test(void);

extern int Flubber;
extern char Str[];
extern int Tester;

// print hello world and the value of Flubber then call Test
void
main()
{
        printf( "Hello, world: %d\n", Flubber );
        Test();
}

// print a message and the values of Str and Tester
void
Test()
{
        printf( "This is a test function\n%s: %d\n", Str, Tester );
}
```

End Listing Six

Listing Seven

```
// DLL.H  Include file for DLL
// 02/12/89 By Gary Syck

#ifdef MAIN
#define EXTERN
#else
#define EXTERN extern
#endif

typedef struct {
        char            *Name;          // Name of the symbol
        unsigned        Size;           // size of the symbol
        unsigned        Seg;            // Segment containing the symbol
        unsigned        Offset;         // Offset for the symbol
} SYMTAB;

#define STUBSIZE 50
typedef struct {
        char            *Name;          // name of the function
        void            *Loc;           // where the function is stored
        int                     Flag;   // true if function is executing
        unsigned char Stub[STUBSIZE];   // Function to call this function
} FUNCTAB;

SYMTAB Syms[200];                       // the symbols found
EXTERN int      SymCnt;                 // The number of symbols
EXTERN int      AllocNumb;              // The first symbol in the current
                                        // allocation block
EXTERN SYMTAB   Syms[];                 // list of symbols
EXTERN int SymCnt;                      // number of symbols
EXTERN unsigned SymSize;                // running total of the size of stuff
EXTERN int      DataSize;               // how many bytes in DataSpace are used

EXTERN FUNCTAB FuncLst[100];            // the list of functions
EXTERN int       FuncCnt;               // The number of allocated entries

extern  unsigned char DataSpace;        // room in the data segment
extern char Stub;                       // This is really a function. It is
                                        // used this way to make sure the
                                        // segment gets used in memcpy
/* record types */
#define THEADR          0x80
#define COMMENT 0x88
#define MODEND          0x8A
#define EXTDEF          0x8C
#define TYPDEF          0x8E
#define PUBDEF          0x90
#define LINNUM          0x94
#define LNAMES          0x96
#define SEGDEF          0x98
#define GRPDEF          0x9A
#define FIXUPP          0x9C
#define LEDATA          0xA0
#define LIDATA          0xA2
#define COMDEF          0xB0

// Function prototypes
void main( int argc, char *argv[] );
void LoadFunc( FUNCTAB *Func );
void GetData(void);
void AllocateSyms(void);
```

End Listings

Listing One *(Text begins on page 40.)*

```
/* Compile with: ZTC wc -ml                                          */
/* (Use large model so strcmp() can handle far pointers.)  */

#include        <stdio.h>
#include        <stdlib.h>
#include        <string.h>
#include        <ctype.h>
#include        <handle.h>

struct tree
    {
        char __handle *word;
        int count;
        struct tree __handle *left;
        struct tree __handle *right;
    };
int readword(char *w, int nbytes)
    {
    int c;
    do  {
        c = getchar();
                if (c == EOF)
                        return 0;
        }
    while (!isalpha(c));
    do  {
        if (nbytes > 1)
            {
            *w++ = c;
                        nbytes--;
            }
        c = getchar();
        }
    while (isalnum(c));
        *w = 0;
        return 1;
    }
void tree_insert(struct tree __handle * __handle *pt, char *w)
    {
        int cmp;
        struct tree __handle *p;
        while ((p = *pt) != NULL)
        {
        if ((cmp = strcmp(w,p->word)) == 0)
            goto gotit;
                pt = (cmp < 0) ? &p->left : &p->right;
        }
    p = (struct tree __handle *) handle_calloc(sizeof(struct tree));
        if (!p || (p->word = handle_strdup(w)) == NULL)
        {
        printf("Out of memory\n");
                exit(EXIT_FAILURE);
        }
    *pt = p;
    gotit:
        p->count++;
    }
tree_print(struct tree __handle *p)
    {
        while (p)
        {
        tree_print(p->left);
        printf("%5d %s\n",p->count,(char far *)p->word);
                p = p->right;
        }
    }
tree_free(struct tree __handle *p)
    {
    struct tree __handle *pn;
        while (p)
        {
        handle_free(p->word);
                tree_free(p->left);
                pn = p->right;
                handle_free(p);
                p = pn;
        }
    }
main()
    {
        struct tree __handle *root = NULL;
        char word[32];
        while (readword(word,sizeof(word)))
        tree_insert(&root, word);
    tree_print(root);
    tree_free(root);
        return EXIT_SUCCESS;
    }
```

End Listing

Listing One *(Text begins on page 58.)*

```
        .
        .
        .
typedef struct {
        int anInt;
        long aLong;
        char tenChars[10];      /* details are unimportant */
} Thing;
typedef union utag {
        Thing aThing;
        union utag *next;
} freeList;
        .
        .
        .
freeList *freeThings = 0;       /* free list, empty to begin */
Thing *newThing()       /* new or recycled Thing */
{
        Thing *t;

        if (freeThings) {       /* any on free list? */
                t = (Thing *) freeThings;
                freeThings = freeThings->next;
        }
        else    /* no, go malloc one */
                t = (Thing *) malloc(sizeof *t);
        return (t);
}
void freeThing(theThing)        /* put Thing on freeList */
Thing *theThing;
{
        ((freeList *) theThing)->next = freeThings;
        freeThings = (freeList *)  theThing;
}
```

End Listing One

Listing Two

```
/* Header for memory control package
 C 1989 Robert A. Moeser, all rights reserved */

/* some things probably in your standard headers... */
#ifndef __size_t
#define __size_t
typedef unsigned long size_t;
#endif

void *malloc(size_t);
void free(void *);
int printf(char *, ...);
int sprintf(char *, char *, ...);
int scanf(char *, ...);
int rand(void);

/* end of things probably in your standard headers... */
typedef long counter;           /* a bit excessive, perhaps... */
typedef struct _mc {
        struct _mc *next;       /* used for doubly-linked list */
        struct _mc *prev;       /* of all mcs */
        size_t itemSize;        /* the size of the object this MCon controls */
        void *freeList;         /* a free list of objects */
        counter nGiven;         /* number of items handed out */
        counter nOut;           /* number still out there somewhere */
        counter nFree;          /* length of free list */
} *MCon;

MCon newMCon(size_t theItemSize);       /* 0 = could not get storage */
void *newInstanceOf(MCon theCon);       /* 0 = could not get storage */
void disposeInstance(void *theItem);
void purgeMCon(MCon theCon);
counter disposeMCon(MCon theCon);    /* number of orphans created by action */
void purgeAllMCons(void);
counter disposeAllMCons(void);       /* number of orphans created by action */
```

End Listing Two

Listing Three

```
/*  A Memory Controller. C 1989 Robert A. Moeser, all rights reserved */

# include "mem.h"

static void linkIn(MCon theCon);
static void linkOut(MCon theCon);
static int initMControl(void);

void debugCon(char *);
void printCon(MCon theCon);

static MCon controlCon = 0;             /* 0 -> not yet initialized */

/* Initialize the memory control "package." Returns 1 for successful
initialization or already initialized; returns 0 if it fails. Called
automatically by newMCon, but you can call it explicity if you like. */

int initMControl(void)
{
        if (controlCon) return (1);             /* already initialized! */
        controlCon = (MCon) malloc(sizeof *controlCon);
        if (!controlCon) return (0);    /* sad but true! */
        controlCon->itemSize = sizeof *controlCon;
        controlCon->nGiven = controlCon->nOut = controlCon->nFree = 0;
        controlCon->freeList = (void *) 0;
        controlCon->next = controlCon->prev = controlCon;
        return (1);                                             /* OK! */
}

/*  Register a new type (object) for management by the memory controller
takes size of the object as an argument and returns an MCon object, which
is zero in case of failure. The Mcon object is used later to manage creation
of all instances of the new object, to maintain a free list for the object
and to keep track of demand for the object */

MCon newMCon(size_t theItemSize)
{
        MCon t;
        int k;
        k = initMControl();
        if (!k) return ((void *) 0);
        t = newInstanceOf(controlCon);
        if (!t) return ((void *) 0);
        t->itemSize = theItemSize;
        t->nGiven = t->nOut = t->nFree = 0;
        t->freeList = (void *) 0;
        linkIn(t);
        return (t);
}

/* Create a new object. Takes the object's MCon as an argument (previously
created by newMCon). Returns a pointer to a new instance of the object */
void *newInstanceOf(MCon theCon)
{
        void *t;
        if (theCon->freeList) {         /* any on free list? */
                t = theCon->freeList;
                theCon->freeList = *((MCon *) t);
                theCon->nFree--;
        }
        else                            /* nope, go malloc one */
                t = malloc(theCon->itemSize + (sizeof theCon));
        if (!t) return ((void *) 0);    /* allocator failure... */
        *((MCon *) t) = theCon;         /* remember the controller */
        theCon->nGiven++;
        theCon->nOut++;
        return ((void *) ((MCon *) t + 1));
}

/* Dispose of an object. Takes a pointer to the object to dispose of
the storage is kept by the object's MCon on a free list for reuse. */
void disposeInstance(void *theItem)
{
        MCon t;
        void *x;
        int g;
        t = *((MCon *) theItem - 1);    /* recover controllor */
        t->nOut--;
        t->nFree++;
        *((MCon *) theItem - 1) = t->freeList;
        t->freeList = (MCon *) theItem - 1;
}

/* Purge the free list for an MCon. Takes the MCon whose free list should be
returned to the system storage allocator */
void purgeMCon(MCon theCon)
{
        void *bop, *bop2;
        bop = theCon->freeList;
        while (bop) {
                bop2 = *((void **) bop);
                free(bop);
                bop = bop2;
        }
        theCon->freeList = (void *) 0;
        theCon->nFree = 0;
}

/* Unregister a type. Takes the MCon object that is no longer needed
returns the number of "orphans" created. Since active instances are entirely
the responsibility of clients of the memory control package, disposing of
an MCon can mean that there may be outstanding objects of the
no-longer-existing type. This is not strictly an error, but since there is
now no way to free the storage used by the orphans should be a warning sign. */

counter disposeMCon(MCon theCon)
{
        counter orphans = 0;    /* did this dispose create "orphans" ? */
        orphans = theCon->nOut;
        purgeMCon(theCon);      /* goodbye the free list */
        linkOut(theCon);
        disposeInstance((void *) theCon);
        return (orphans);       /* number of "orphans", should be 0! */
}

/* Purge the free list of every object type known to the memory controller
a client might call this in a desperate attempt to satisfy a memory request */
void purgeAllMCons(void)
{
        MCon bop;
        bop = controlCon;
        do {
                purgeMCon(bop);
                bop = bop->next;
        } while (bop != controlCon);
}

/* Unregister all types. This action can create orphans in the same sense as
disposeMCon above, and so returns the number. It really ought to be zero
unless the client has created objects meant to endure until program
termination. Since all types are deactivated the additional step of
deactivating the memory controller itself is taken. All storage used by the
package is returned to the system storage allocator. */

counter disposeAllMCons(void)
{
        counter orphans = 0;
        MCon bop, bop2;
```

(Listing continued on page 112)

Listing Three *(Listing continued, text begins on page 58.)*

```
        bop = controlCon->next;
        while (bop != controlCon) {
                bop2 = bop->next;
                orphans += disposeMCon(bop);
                bop = bop2;
        }
        purgeMCon(controlCon);
        free((void *) controlCon);
        controlCon = (void *) 0;        /* de-initialize mc */
        return (orphans);       /* total number of "orphans", should be 0! */
}

/* Internal Routines */

/* Link a new MCon into the doubly-linked list of all known MCons
the list head and tail is the MCon for MCons, so there is always at least one
item on the list and the first item is always the MCon's MCon */
static void linkIn(MCon theConToAdd)
{
        theConToAdd->next = controlCon->next;
        theConToAdd->prev = controlCon;
        controlCon->next = theConToAdd;
        (theConToAdd->next)->prev = theConToAdd;
}

/* Unlink an MCon (called upon destruction of an Mcon */
static void linkOut(MCon theConToDel)
{
        (theConToDel->prev)->next = theConToDel->next;
        (theConToDel->next)->prev = theConToDel->prev;
}

/* Debugging and Utility Routines */

/* Print a list of known MCons with statistics. Takes a tag to accompany
printout as an argument. */
void debugCon(char *s)
{
        MCon bop;
        printf("%s", s);
        if (!controlCon) {
                printf(" -mc is OFF!\n");
                return;
        }
        bop = controlCon;
        do {
                printCon(bop);
                bop = bop->next;
        } while (bop != controlCon);
}

/* Print one MCon. Make a quick check on the integrity of the internal
data structure. */
void printCon(MCon theCon)
{
        char *freeBop;
        int freeCount;
        int OK;
        freeBop = theCon->freeList;
        freeCount = 0;
        while (freeBop) {
                freeCount++;
                freeBop = *((char **) freeBop);
        }
        OK = freeCount == theCon->nFree;
        printf("%lx = %lu\t%ld\t%ld\t%ld\t%lx\t%s\n",
                theCon,
                theCon->itemSize,
                theCon->nGiven,
                theCon->nOut,
                theCon->nFree,
                theCon->freeList,
                OK ? "<ok>" : "<NOK>");
}
```

End Listing Three

Listing Four

```
/* Part 1 of torture-test of the memory controller */

# include <stdio.h>
# include <console.h>

void torture(void);
main()
{
        int i;
        for (i = 0; i < 100; i++)
                torture();
}
```

End Listing Four

Listing Five

```
/* Part 2 of torture-test of the memory controller */

# include "mem.h"

# define MAXTYPES               20
# define MAXINSTANCES   500
# define BASESIZE               25
# define MAXEXTRA               25
# define NPASSES                50000

void torture(void);
void error(char *);
void makeNewType(void);
void makeNewObject(void);
void freeSomeObject(void);
int randle(int);
void debugCon(char *);
struct {
        MCon mCon;
        long theID;
} regType[MAXTYPES];
struct {
        void *mObj;
        long typeID;
} mBag[MAXINSTANCES];
int nInstances = 0;
int nTypes = 0;
long serialID = 1000;
void torture()
{
        counter orphans = 0;
        size_t sizeItem;
        int typeIdx, objIdx;
        int nPotential, nActual;
        long tID;
        int i, x;
        long looper;
        char msg[64];
        for (looper = 0; looper < NPASSES; looper++) {
                if (orphans) error("orphans have been created");
                if (!nTypes) {
                        if (nInstances) error("instances but no active types");
                        makeNewType();
                }
                x = randle(10);
                switch (x) {
                case 0 :                                /* make a report */
                        sprintf(msg, "loop %ld : %d instances of %d types\n",
                          looper,
                          nInstances,
                          nTypes);
                        debugCon(msg);
                        break;
                case 1 :                        /* register a new type if room */
                        makeNewType();
                        break;
                case 2 :                        /* make a new object if room */
                        makeNewObject();
                        break;
                case 7 :                        /* make many new objects */
                        nPotential = ((MAXINSTANCES - nInstances) >> 2) + 7;
                        nActual = randle(nPotential);
                        while (nPotential--)
                                makeNewObject();
                        break;
                case 3 :                        /* free an object if any exist */
                        freeSomeObject();
                        break;
                case 8 :                        /* free many objects */
                        nPotential = ((MAXINSTANCES - nInstances) >> 3) + 3;
                        nActual = randle(nPotential);
```

```
                while (nPotential--)
                        freeSomeObject();
                break;
        case 4 :                /* purge free list of a type if any exist */
                if (nTypes) {
                        typeIdx = randle(nTypes);
                        purgeMCon(regType[typeIdx].mCon);
                }
                break;
        case 9 :                        /* purge a number of free lists */
                if (nTypes) {
                        nActual = randle(nTypes);
                        for (i = 0; i < nActual; i++) {
                                typeIdx = randle(nTypes);
                                purgeMCon(regType[typeIdx].mCon);
                        }
                }
                break;
        case 5 :        /* free all of a registered type if any exist */
                if (nTypes) {
                        typeIdx = randle(nTypes);
                        tID = regType[typeIdx].theID;
                        for (i = 0; i < nInstances; )
                                if (mBag[i].typeID == tID) {
                                        disposeInstance(mBag[i].mObj);
                                        nInstances--;
                                        mBag[i] = mBag[nInstances];
                                }
                                else i++;
                                /* and maybe kill the controllor */
                        if (randle(13) > 7) {
                         orphans += disposeMCon(regType[typeIdx].mCon);
                         nTypes--;
                         regType[typeIdx] = regType[nTypes];
                        }
                }
                break;
        case 6 :                        /* free all instances of all types */
                if (randle(20) < 15) break;
                for (i = 0; i < nInstances; i++)
                        disposeInstance(mBag[i].mObj);
                nInstances = 0;
                                /* kill some controllors */
                for (i = 0; i < nTypes;)
                        if (randle(15) > 13) {
                            orphans += disposeMCon(regType[i].mCon);
                            nTypes--;
                            regType[i] = regType[nTypes];
                        }
                        else i++;
                   /* and maybe kill all the rest and shut down! */
                if (randle(20) > 17) {
                        orphans += disposeAllMCons();
                        nTypes = 0;
                }
                break;
        }
    }
    printf("\n%ld passes...\n", NPASSES);
}
void makeNewType()
{
    MCon tCon;
    size_t sizeItem;
    if (nTypes < MAXTYPES) {
        sizeItem = BASESIZE + randle(MAXEXTRA);
        tCon = newMCon(sizeItem);
        if (!tCon) {
            error("could not make controllor");
            return;
        }
        regType[nTypes].mCon = tCon;
        regType[nTypes].theID = serialID++;
        nTypes++;
    }
}
void makeNewObject()
{
    int typeIdx;
    void *tObj;
    if ((nInstances < MAXINSTANCES) && nTypes) {
        typeIdx = randle(nTypes);
        tObj = newInstanceOf(regType[typeIdx].mCon);
        if (!tObj) {
            error("could not get object memory");
            return;
        }
        mBag[nInstances].mObj = tObj;
        mBag[nInstances].typeID = regType[typeIdx].theID;
        nInstances++;
    }
}
void freeSomeObject()
{
    int objIdx;
    if (nInstances) {
        objIdx = randle(nInstances);
        disposeInstance(mBag[objIdx].mObj);
        nInstances--;
        mBag[objIdx] = mBag[nInstances];
    }
}
void error(char *s)                     /* print error message and hang */
{
    char hang[12];
    printf("\nERROR : %s\n", s);
    scanf("%s", hang);
}
randle(int n)                           /* random number up to not including n */
{
    if (n <= 0) error("bogus randle call");
    return (rand() % n);
}
```

End Listings

Listing One *(Text begins on page 82.)*

```
Object subclass: #RandomNumber
  instanceVariableNames: ''
  classVariableNames: 'Seed'
  poolDictionaries: ''.

"RandomNumber class methods"
new
    |n|
    Seed isNil ifTrue:[self reset].
    n := Seed. Seed := (Seed * 263 + 30011) bitAnd:16r7FFF.
    ^n.
from:min to:max
    ^(self new \\ (max - min + 1)) + min.
randomize
    Seed := Time millisecondClockValue bitAnd:16r7FFF.
reset
    Seed := 0.

Object subclass: #MarketActor
  instanceVariableNames: 'running msgQueue position notify'
  classVariableNames: ''
  poolDictionaries: ''.

"MarketActor class methods"
new
    ^super new initialize.
priority
    ^2.
imageName
    ^'PersonUp'.
formNamed:anImageName
    ^MarketImages at:anImageName.
form
    ^self formNamed:self imageName.
extent
    ^self form extent.
height
    ^self form height.
width
    ^self form width.

"MarketActor instance methods"
initialize
    msgQueue := MessageQueue new.  running := false.
release
    msgQueue release.  msgQueue := nil.  notify := nil.
    super release.
alternateForm
    ^self formNamed:self class alternateImageName.
display
    self display:self form at:position.
display:aForm
    self display:aForm at:position.
display:aForm at:aPoint
    aForm displayAt:aPoint.
animate
    self display:self alternateForm.
erase
    Display white:self frame.
form
    ^self class form.
formNamed:anImageName
    ^self class formNamed:anImageName.
frame
    ^position extent:self extent.
extent
    ^self class extent.
height
    ^self class height.
width
    ^self class width.
messageQueue
    ^msgQueue.
receive
    ^msgQueue receive.
send:aMessage to:anObject
    self send:aMessage to:anObject with:nil.
send:aMessage to:anObject with:anArgument
    |queue|
    (queue := anObject messageQueue) isNil
        ifFalse:[
            queue send:aMessage.
            anArgument isNil
            ifFalse:[queue send:anArgument]].
    Processor yield.
position:aPoint
    position := aPoint.
moveBy:delta
    self moveTo:position + delta.
moveTo:aPoint
    self erase;  position:aPoint;  display.
start
    running
        ifFalse:[
            running := true.
            [self run] forkAt:self class priority].
run
    |action|
    [running]
        whileTrue:[
            action := self receive.
            (running and:[self respondsTo:action])
                ifTrue:[self perform:action]].
    self shutdown.
stop:notifySemaphore
    notify := notifySemaphore.
    running
        ifTrue:[
            running := false.
            self send:#wakeUp to:self]
        ifFalse:[self notify].
shutdown
    self erase;  notify.
notify
    (notify isKindOf:Semaphore) ifTrue:[notify signal].
sleep:numberOfSeconds
    |timeout|
    timeout := Time now asSeconds + numberOfSeconds.
    [running and:[Time now asSeconds < timeout]]
        whileTrue:[Processor yield].
wakeUp
        "Do nothing -- in case the receiver
         is waiting at the message queue."

MarketActor subclass: #Bagger
  instanceVariableNames: 'checker'
  classVariableNames: ''
  poolDictionaries: ''.

"Bagger class methods"
alternateImageName
    ^'PersonRight'.

"Bagger instance methods"
release
    checker := nil.  super release.
checker:aChecker
    checker := aChecker.
takeItem
    running
        ifTrue:[
            self
                animate;
                sleep:1;
                send:#gotIt to:checker;
                display].

MarketActor subclass: #Checker
  instanceVariableNames: 'bagger customer'
  classVariableNames: ''
  poolDictionaries: ''.

"Checker class methods"
alternateImageName
    ^'PersonLeft'.

"Checker instance methods"
release
    bagger := nil.  customer := nil.  super release.
bagger:aBagger
    bagger := aBagger.
checkOutCustomer
    customer := self receive.
gotIt
    running
        ifTrue:[
            self
                display;
                send:#removeItemFromCart to:customer].
takeItem
    running
        ifTrue:[
            self
                animate;
                send:#takeItem to:bagger;
                sleep:1;
                send:#gotIt to:customer].

MarketActor subclass: #CheckoutCounter
  instanceVariableNames: 'checker bagger customers nowServing'
  classVariableNames: 'MaxCustomers'
  poolDictionaries: ''.

"CheckoutCounter class methods"
initialize
    MaxCustomers isNil ifTrue:[MaxCustomers := 3].
new
    self initialize.  ^super new.
imageName
    ^'Counter'.
maxCustomers
    self initialize.  ^MaxCustomers.
maxCustomers:aNumber
    MaxCustomers := aNumber.
priority
    ^3.

"CheckoutCounter instance methods"
initialize
    checker := Checker new.  bagger  := Bagger new.
    checker bagger:bagger.   bagger  checker:checker.
    customers := OrderedCollection new:MaxCustomers.
    super initialize.
release
    bagger release.  checker release.  nowServing release.
    bagger := nil.   checker := nil.   nowServing := nil.
    [customers isEmpty]
        whileFalse:[customers removeFirst release].
    customers release.  customers := nil.  super release.
position:aPoint
    super position:aPoint.
    checker position:aPoint
                        + ((self width // 2)
                            @ (self height // 4)).
    bagger position:aPoint + (0 @ self height * 3 // 4).
checkoutPosition
    ^position - (Customer width @ 0).
endOfLinePosition
```

(Listing continued on page 118)

Listing One *(Listing continued, text begins on page 82.)*

```
    ^position
        - (Customer width
            @ (customers size + 1 * Customer height)).
exitPosition
    ^position + ((0 - Customer width) @ self height).
length
    ^customers size
        + (nowServing isNil ifTrue:[0] ifFalse:[1]).
display
    super display.  checker display.  bagger display.
addCustomer
    |aCustomer|
    aCustomer := self receive.
    running
        ifTrue:[
            customers add:aCustomer.
            self length == 1
                ifTrue:[self nextCustomer]].
nextCustomer
    running ifTrue:[^self].
    customers isEmpty
        ifTrue: [nowServing := nil]
        ifFalse:[
            nowServing := customers removeFirst.
            self send:#moveToCheckout
                    to:nowServing with:checker.
            customers do:[:aCustomer|
                aCustomer isNil
                    ifFalse:[self send:#moveForward
                                    to:aCustomer]]]].
start
    super start.  bagger start.  checker start.
shutdown
    |semaphore|
    self update:0.  semaphore := Semaphore new.
    customers do:[:aCustomer|
            aCustomer stop:semaphore.  semaphore wait].
    nowServing isNil
        ifFalse:[nowServing stop:semaphore.  semaphore wait].
    checker isNil
        ifFalse:[checker stop:semaphore.  semaphore wait].
    bagger isNil
        ifFalse:[bagger stop:semaphore.  semaphore wait].
    super shutdown.
update
    |aValue|
    ((aValue := self receive) isKindOf:Number)
        ifTrue:[self update:aValue].
update:aValue
    |fieldWidth textPosition extent|
    textPosition :=
        position + ((self width // 2)
                    @ ((SysFont height + 2) negated)).
    fieldWidth := 3.
    extent := (SysFont width * fieldWidth)
            @ SysFont height.
    aValue == 0
        ifTrue: [Display white:(textPosition extent:extent)]
        ifFalse:[(aValue printString flushedRightIn:fieldWidth)
                    displayAt:textPosition].

MarketActor subclass: #Customer
  instanceVariableNames: 'counter checker items leaving'
  classVariableNames: 'MinItems MaxItems'
  poolDictionaries: ''.

"Customer class methods"
alternateImageName
    ^'CustomerReaching'.
imageName
    ^'Customer'.
new
    MinItems isNil ifTrue:[MinItems := 5.  MaxItems := 25].
    ^super new.

"Customer instance methods"
initialize
    items := RandomNumber from:MinItems to:MaxItems.
    leaving := false.  super initialize.
release
    checker := nil.  counter := nil.  super release.
counter:aCheckoutCounter
    counter := aCheckoutCounter.
cartIsEmpty
    ^items <= 0.
moveToEndOfLine
    running ifTrue:[self moveTo:counter endOfLinePosition].
moveForward
    running ifTrue:[self moveBy:0 @ self height].
moveToCheckout
    checker := self receive.
    running
        ifTrue:[
            self
                moveTo:counter checkoutPosition;
                send:#checkOutCustomer to:checker with:self;
                update;
                removeItemFromCart].
leaveStore
    running
        ifTrue:[
            self
                update;
                moveTo:counter exitPosition;
                send:#nextCustomer to:counter.
            running := false.  leaving := true].
gotIt
    self display;  sleep:1.
removeItemFromCart
    running
        ifTrue:[
            self cartIsEmpty
                ifTrue:[self leaveStore]
                ifFalse:[
                    self
                        update;
                        animate;
                        send:#takeItem to:checker.
                    items := items - 1]].
shutdown
    super shutdown.  leaving ifTrue:[self release].
update
    (running and:[counter notNil])
        ifTrue:[self send:#update to:counter with:items].
```

End Listing One

Listing Two

```
Object subclass: #MarketSimulator
  instanceVariableNames:
    'running notify counters frame statusFrame
     totalCustomers startTime'
  classVariableNames: 'MaxTime MaxCounters MinTime '
  poolDictionaries: ''.

"MarketSimulator class methods"
new
    MaxCounters := 3.  MinTime := 1.  MaxTime := 15.
    ^super new initialize.
priority
    ^2.

"MarketSimulator instance methods"
initialize
    totalCustomers := 0.  running := false.
release
    1 to:counters size do:[:i|
        (counters at:i) release.
        counters at:i put:nil].
    counters release.  counters := nil.
    notify := nil.     super release.
reframe:aFrame
    |statusHeight w h maxCounters maxCustomers x y|
    CursorManager execute change.
    frame := aFrame.
    statusHeight := SysFont height + 4.
    statusFrame := aFrame origin + (2 @ 2)
                    extent:(aFrame width - 4) @ statusHeight.
    w := CheckoutCounter width + Customer width + 2.
    h := CheckoutCounter height + Customer height + 4.
    maxCounters := ((aFrame width - Customer width) // w)
                    min:MaxCounters.
    maxCustomers := ((aFrame height - statusHeight - h)
                    // Customer height)
                        min:CheckoutCounter maxCustomers.
    x  := aFrame origin x
            + ((aFrame width - (maxCounters * w)) // 2)
            + Customer width.
    y  := aFrame corner y - h.
    CheckoutCounter maxCustomers:maxCustomers.
    counters := Array new:maxCounters.
    1 to:maxCounters do:[:i|
        counters
            at:i put:(CheckoutCounter new
                        position:x @ y;
                        display;
                        yourself).
        x := x + w].
    CursorManager normal change.
send:aMessage to:anObject
    self send:aMessage to:anObject with:nil.
send:aMessage to:anObject with:anArgument
    |queue|
    (queue := anObject messageQueue) isNil
        ifFalse:[
            queue send:aMessage.
            anArgument isNil ifFalse:[queue send:anArgument]].
    Processor yield.
sleep:numberOfSeconds
    |timeout lastTime time|
    timeout := Time now asSeconds + numberOfSeconds.
    lastTime := 0.
    [running and:[(time := Time now asSeconds) < timeout]]
        whileTrue:[
            time = lastTime
                ifFalse:[
                    self
                        timeRemaining:(timeout - time);
                        elapsedTime.
                    lastTime := time].
            Processor yield].
shortestLine
    |fewest length shortest|
    fewest := 9999.
    1 to:counters size do:[:i|
        (length := (counters at:i) length) < fewest
            ifTrue:[fewest := length. shortest := i]].
    fewest < CheckoutCounter maxCustomers
```

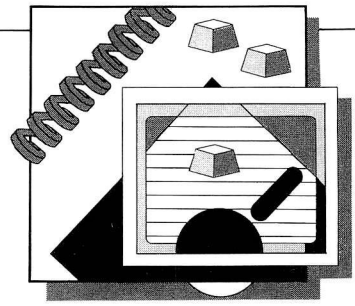
```
        ifTrue:[^counters at:shortest]
        ifFalse:[^nil].
newCustomer
    |counter customer|
    (counter := self shortestLine) isNil
        ifFalse:[
            customer := Customer new
                counter:counter;
                position:counter endOfLinePosition;
                display;
                start;
                yourself.
            totalCustomers := totalCustomers + 1.
            ('CUSTOMERS SERVED:', totalCustomers printString)
                displayAt:statusFrame origin.
            self send:#addCustomer to:counter with:customer].
start
    counters do:[:aCounter|
        aCounter isNil ifFalse:[aCounter start]].
    running
        ifFalse:[
            startTime := Time now.
            running := true.
            [self run] forkAt:self class priority].
run
    self newCustomer.
    [running]
        whileTrue:[
            self sleep:(RandomNumber from:MinTime to:MaxTime).
            running ifTrue:[self newCustomer]].
    self shutdown.
    (notify isKindOf:Semaphore)
        ifTrue:[notify signal].
running
    ^running.
stop:notifySemaphore
    running
        ifTrue:[
            notify := notifySemaphore.
            running := false]
        ifFalse:[
            self shutdown.
            notifySemaphore signal].
shutdown
    |semaphore|
    semaphore := Semaphore new.
    counters do:[:aCounter|
        aCounter isNil
            ifFalse:[
                aCounter stop:semaphore.
                semaphore wait]].
elapsedTime
    |time field offset|
    time := Time now subtractTime:startTime.
    field := 'ELAPSED TIME ', time printString.
    offset := statusFrame center x
            - ((SysFont stringWidth:field) // 2).
    field displayAt:statusFrame origin + (offset @ 0).
timeRemaining:seconds
    |fieldWidth field offset|
    fieldWidth := 3.
    field := 'NEXT CUSTOMER:',
            (seconds printString flushedRightIn:fieldWidth).
    offset := statusFrame width
            - (SysFont stringWidth:field).
    field displayAt:statusFrame origin + (offset @ 0).

Object subclass: #MarketWindow
  instanceVariableNames: 'topPane aPane simulator'
  classVariableNames: 'Version Frame Title'
  poolDictionaries: ''.

"MarketWindow class methods"
initialize
    Frame   := Display boundingBox insetBy:16.
    Title   := 'Market Checkout Simulation'.
    Version := ' (Version 1.0 -- 02/24/90 -- KEA)'.
new
    self initialize.
    ^super new.
"MarketWindow instance methods"
open
    topPane := TopPane new
        model:self;
        label:Title, Version;
        menu:#windowMenu;
        rightIcons:#(collapse);
        yourself.
    topPane addSubpane:
        (aPane := GraphPane new
            model:self;
            name:#initPane:;
            menu:#paneMenu;
            framingRatio:(0 @ 0 extent:1 @ 1)).
    topPane reframe:Frame.
    topPane dispatcher openWindow scheduleWindow.
close
    self stop.
initPane:aFrame
    Display white:aFrame.
    self initSimulator:aFrame.
    ^Form fromDisplay:aFrame.
initSimulator:aFrame
    simulator isNil
        ifTrue:[
            simulator := MarketSimulator new
                reframe:aFrame;
                yourself].
```

(continued on page 120)

Listing Two *(Listing continued, text begins on page 82.)*

```
paneMenu
    (simulator isNil or:[simulator running])
        ifTrue:[^EmptyMenu new]
        ifFalse:[^Menu
                    labels:'START SIMULATION'
                    lines:#()
                    selectors:#(start)].
windowMenu
    ^Menu
        labels:'cycle\collapse\close' withCrs
        lines:#()
        selectors:#(cycle collapse closeIt).
start
    simulator start.
stop
    |semaphore|
    simulator isNil
        ifFalse:[
            CursorManager execute change.
            semaphore := Semaphore new.
            simulator stop:semaphore.
            semaphore wait;  release.
            simulator release.  simulator := nil.
            CursorManager normal change].
```

End Listings



Complex Systems, Fractals, and Chaos

Michael Swaine

Too bad complexity isn't a little simpler. Last month I presented several of the current views on how to manage complex systems. These were all software engineering strategies for getting organized, the idea being that we need lots of order if we hope to cope with complexity in unpredictable environments.

This month chaos gets its turn.

There is mounting evidence that the management of a complex system in an uncertain environment requires a healthy dose of chaos. Some of that evidence was presented in a recent *Scientific American* article, "Chaos and Fractals in Human Physiology," by Ary L. Goldberger, David R. Rigney, and Bruce J. West, (February, 1990). Goldberger et al focus on the management of one of the most complex of systems, the human body, but their conclusions are interesting for what they say about complex systems in general. There is nothing inherently biological in their mathematics.

Every operating system designer knows you need a little randomness to avoid certain problems. The Dining Philosophers problem, discussed here and more fully in David Harel's book, *Algorithmics* (Addison-Wesley, 1987), is a classic case of deadlock that cannot be resolved without introducing an element of randomness. Any strictly deterministic solution to the Dining Philosophers, problem is guaranteed to fail. While this sounds like what Goldberger et al are saying about chaos in human physiology, there is an important difference. Goldberger et al are not merely describing a useful random tweaking of an existing model, but throwing out an existing mathematical model and replacing it with another.

That new model is deterministic chaos, and that phrase is not, apparently, an oxymoron. (My favorite recent additions to the oxymoron lexicon are "Justice Rehnquist," thanks to John Perry Barlow, retired cattle rancher, Grateful Dead lyricist, and computer book author; and "reputable astrologer," an oxymoronic self-characterization by a notorious Reagan family advisor.)

Deterministic chaos arises from the discipline of nonlinear dynamics, which is the study of systems that respond disproportionately to stimuli. There are certain situations in which nonlinear systems that are strictly deterministic nevertheless behave in seemingly random ways. This is not true randomness, but a constrained but erratic behavior called "chaos." It is the chaos of the heart.

Times of the Heart

The conventional physiological model that covers health, disease, and aging is homeostasis. According to this model, physiological systems act to reduce variability and to maintain constant or regularly periodic internal states. Fluctuations in heart rate have been viewed as the response to external stresses, and it has been assumed that the physiological system functions to return the heart rate to its normal level. A perfectly normal heart in a perfectly stressless environment would, under this model, tick like a metronome.

The authors argue that the conventional model is wrong. Within the past five years evidence has been accumulating that chaotic behavior in physiological systems may be the product of healthy functioning, while regular, periodic behavior or steady-state behavior is symptomatic of some problem. Healthy young hearts often exhibit the greatest irregularity, while regularity of heartbeat is sometimes a precursor of heart failure.

One study the authors cite depends on examining heart rate data plotted as Fourier spectra and phase-space plots. Fourier analysis displays periodicity clearly as spikes; steady-state behavior as low, flat lines; and chaotic behavior as a broad spectrum. Phase-space plots give a different picture, but are even better at identifying chaotic behavior. In phase-space plots, periodicity shows up as clearly cyclic figures called "limit cycles," steady-state behavior maps into a point, and chaotic behavior produces something called a "strange attractor," which looks clearly chaotic.

The results they cite are impressive. A healthy heart produced a broad Fourier spectrum and a strange attractor. Unhealthy hearts showed either a Fourier spike and a limit cycle, or a nearly flat Fourier spectrum and a point attractor. Chaos, apparently, is healthy.

This chaos seems to be directed from the nervous system, where researchers are finding evidence of further chaos. Significantly, heart-rate variability decreases after a heart transplant, which requires severing a connection between the heart and the nervous system. Chaos also seems to be present in the nervous control of hormonal secretion and in other areas in the nervous system. One model shows how chaos could be produced in the nervous system: It involves feedback loops among neurons.

There's another chaotic clue in the nervous system. The branching structure of neurons seems to have fractal dimension, which is significant because of the connection between fractals and chaos.

River's Edge

Just what the relationship between chaos and fractals is is not entirely clear. But there is a connection: Fractals are often the remnant of chaotic nonlinear dynamics. The picture seems to be this: A chaotic process shapes an environment, and the trace left behind is a fractal.

A fractal, then, is a geometric form with the following distinctive characteristics: Infinite detail and self-similarity at any scale. No matter how closely you examine it, you find more detail, and it looks more or less the same at any level of magnification. A fractal can be an infinitely branching line, an infinitely lumpy surface, or any similarly hairy object of higher dimensionality.

The dimensionality of a fractal is more complicated than this, though. Because of its infinite detail, a fractal does not really have a dimension in the conventional sense. An infinitely branching line has no single measure of length, and is not a one-dimensional object. A fractal's dimension is defined to be a

function of the probability that it touches any given point in the space containing the fractal. The dimension of an infinitely branching line fractal is a number between 1 and 2. This fractional dimensionality is where fractals get their name. There are some dizzying consequences of this fractional dimensionality, such as the ideas that a coastline doesn't have any definable length, and that a river has no edge.

Fractals have received a lot of attention in computer science since their discovery by Benoit Mandelbrot. But they are not just of academic interest. In cinematic computer graphics and elsewhere, fractals are proving to be powerful. If you want to model branching anatomical structures such as lungs and nerves, flora-like trees or shrubbery, or coastlines or river meanders or mountain chains or any sort of terrain, you will do well to examine fractals.

There have been a number of articles and books about fractals in nature, reminiscent of past articles and books on mathematics in nature. Mathematical functions and forms such as the spiral and helix, and the phi function and Fibonacci series, crop up with amazing frequency in natural forms. Books such as H.E. Huntley's *The Divine Proportion* and Theodore A. Cook's *The Curves of Life*, popular treatments both available in Dover paperback editions, describe how such mathematical entities appear in odd places in nature.

The Fibonacci sequence is particularly common in nature. The sequence begins 1,1 and each succeeding term is the sum of the two that precede it. The pattern of interlocking spirals on the face of a sunflower, with Fib(n) spirals twisting clockwise and Fib(n+1) spirals twisting counter clockwise, the two sets of spirals defined on the same set of florets, is particularly impressive. For sunflowers, the value of n is 8, producing interlocking spirals of 21 and 34 florets. The same mathematic structure appears in other plants: Pinecones and pineapples show the same spirals, but with a different value for n.

Intriguing as these examples of mathematics in nature are, they don't seem to offer any deep insights into nature. Apparently the number of florets in each row outward constrains the number of florets in the next concentric row in a way consistent with the simple rule for generating Fibonacci numbers. Something like that. In any case, the math may be nifty, but the underlying natural process generally turns out on examination not to be profound.

With fractals, there seems to be something deeper at work.

My Science Project

Since reading *The Science of Fractal Images* by Heinz-Otto Peitgen and Dietmar Saupe (Springer-Verlag, 1988), I've been playing with one particular algorithm for producing graphic forms that branch like plants. A program that I've written uses simple transformational rules to produce branching structures that look more like trees than anything I could ever produce by hand.

What I find more interesting, and what has kept me fiddling with the algorithm off and on for two years, is its apparent relevance to the process of growth. It's fascinating to watch the process of development of these fractal flora. Starting from a single shoot, the graphic develops into a twig with a couple of offshoots, then into something that looks like a b-tree in a windstorm, finally turning into a credible sketch of a bare tree. While there exist algorithms for putting leaves on the branches, it's not the verisimilitude of the static image that impresses me, but the accuracy of the simulation of the development process.

Note: The development of the fractal, the series of stages it goes through as it increases in complexity, is strictly an artifact of the way fractals are defined. Although it is possible to pick parameters so as to create a final image resembling one plant or another, it isn't possible, so far as I can tell, to control the process of development by the choice of parameters. I've made no attempt, in any case, to mirror any kind of natural process; the program does it — er, naturally.

Something is going on in the development of the fractal that has something deep in common with what goes on in the growth of a plant. Fractals know something about biological growth.

Perhaps it has something to do with what the *Scientific American* authors say about fractals: That they are often the remnant of chaotic nonlinear dynamics. Apparently the presence in an object of a static structure well modeled by a fractal is some evidence of chaotic nonlinear dynamic processes at work in the development of the object. If that's what's happening in my fractal flora, then the algorithm is doing more than drawing nice tree-like pictures. It is a fairly deep simulation of the process of growth in organisms.

I don't want to overstate the point. I don't think we're on the verge of algorithms for simulating human development that will acquire and lose their

(continued from page 124)
gill slits as ontogeny recapitulates phylogeny. But it is intriguing that the fractal flora simulate stages of natural growth with no prompting in the form of rigged parameters. The fact that fractals have infinite detail and self-similarity implies a lot about how they develop, and in fact allows a very simple initial rule to apply at successively more complex stages of development, just as the rules for organic development must apply to the early stages of organ development and also to the later stages in which organs interact in complex systems.

Fractals and DNA appear to have similar problems to solve. Is it possible that they solve them in similar ways?

Field of Dreams

One of the dreams of science fiction, and consequently of the artificial intelligence community that reads science fiction stories for research topics, has always been the system that programs itself. The machine that actively seeks out knowledge in order to grow more wise. The vague notions of how this might come about seem always to rest on faith in critical mass. Even Douglas Hofstadter's vision of artificial consciousness assumes that sufficient complexity somehow magically transforms a system into an intelligence.

Critical mass probably isn't enough; natural systems need a plan of development, the genetic blueprint. It seems reasonable to expect that highly complex and adaptive artificial systems would need some plan, too. The current Most Likely to Succeed paradigm for machine learning, or adaptive systems, is neural nets. Currently, neural nets are designed with as much naivete regarding neurophysiology as regarding neuroanatomy. There is little reason to think that exposing a blank slate neural net to unpredictable events will lead it somehow to cope with its environment. If neural nets are to grow more complex in useful ways, don't

we need to build in a plan for recognizing what is useful, and shouldn't it be a plan that can admit of more sophisticated interpretation as the system gets more sophisticated?

Do artificial neural nets, such as real networks of neurons, need a dose of chaos?

DDJ

Fractal Flora

Here's a sketch of the fractal program I've written. It doesn't merit a pseudocode description, because the underlying algorithm is not efficient. My purposes had as much to do with teaching HyperTalk as with exploring fractals, so I implemented it so as to keep the concepts visible, employ simple user-comprehensible graphic tools, and use only HyperTalk code. A serious exploration of fractals would have to abandon all of these constraints.

The program uses turtle graphics, which is to say that the user describes the figures to be drawn in terms of strings of one-character commands, which specify the direction and movement of an imaginary drawing turtle. (When you've been editor of a magazine originally called *Dr. Dobb's Journal of Tiny BASIC Calisthenics and Orthodontia*, you learn how to write things like that with a straight face.) The program draws the fractals by passing the turtle graphic commands to a simple turtle graphic engine.

The program recognizes these turtle graphic commands:

Drawing: F Forward 1 unit, pen down; U Forward 1 unit, pen up
Orientation: <Turn one unit left; >Turn one unit right
Context: [Store current turtle position & direction;] Reset turtle to previous position & direction

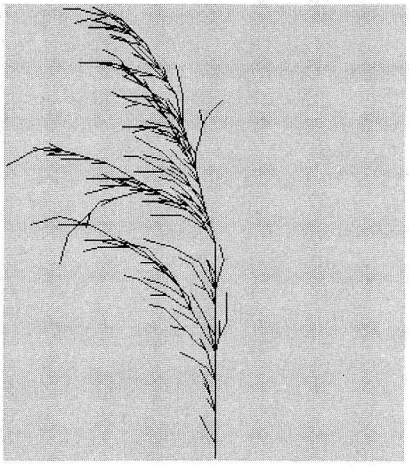


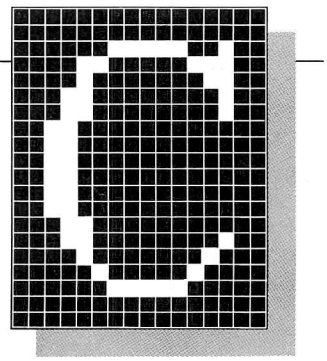
Figure 1: *Typical fractal flora*

Fractals start from a simple base figure, and are transformed to new levels of complexity via transformation rules. The program steps through these levels, transforming the current string of turtle graphic characters into a new string. It draws the current version of the fractal from the turtle graphic string at each level. The "true" fractal requires infinite levels of detail, so the program is only drawing successive approximations. Figure 1 shows a typical image generated by the program.

The program starts by prompting the user for parameters. The user must give the fractal a title, a base string of turtle graphic commands (the single command F is typical), a repeatMode, a unitAngle, and a set of rules for transforming the base string to produce higher levels of the fractal. RepeatMode controls how the program steps through the levels, and unitAngle (0-360) controls how sharply the turtle turns.

A typical transformation rule is F -> F[>F][<F]. This turns each straight line at level n into a fork consisting of one step forward and branches to the left and right. The user enters these transformation rules in response to specific prompts. Each such prompt shows a turtle graphic character and asks what it is to be transformed into. Initially, these characters are just the characters in the base string, but as new rules are added, they may add new characters, requiring new rules. Rules not involving commands [and] will produce figures such as coastlines and mountain ridges; using [and] will produce branching structures such as trees and blood vessels.

Perhaps the main point of describing this process is to show how inefficient fractals are. Because each transformation is a function, one could, I suppose, decide how many levels deep you want to go and compute the composition of the functions needed, applying this function. For me, computing and drawing each successive level is important because it is the process of fractal growth that I'm interested in. — M.S.



SD '90 and ANSI at Last

Al Stevens

This column comes to you from the Software Development '90 conference at Oakland, California. Every year, the conference gets bigger and better. I am honored this year to have been invited to participate on the Comparative Language Panel representing C and C++. More about that later.

SD '90 is the place where vendors of programmers' products — compilers, function libraries, CASE tools, editors, debuggers, and so on — show their stuff to the programmers who attend. Besides the product exhibits, there are workshops and lectures all week long.

The highlight of the show was a two-day session with Bjarne Stroustrup. The creator of C++ gave a wall-to-wall discussion of how to make the transition to his creation, the language that will, I believe, prevail in the '90s.

Turbo Debugger and Tools

Borland surprised us all in a press conference on Tuesday. They announced their new Turbo Debugger and Tools product, but that wasn't the surprise. The first jolt came while we sat waiting for the show to start. The crowd was moving past my seat when I realized that a German Shepherd had her head in my lap. It was Duchess, Debe Norling's seeing eye dog, patiently waiting for Debe to find a place to sit. Debe is a C freelance programmer and writer who has kindly reviewed some of my books in other publications.

The purpose of the press conference was to show some new stuff from Borland. Eugene Wong demonstrated the new Turbo Debugger, Profiler, and Assembler. He interspersed the dazzling demonstration with the expected number of cheap shots at Microsoft accompanied by the usual approving laughter from the audience. The big surprise came at the conclusion of the presentation. Eugene began to tell us about a nifty optimizing C compiler that works with the Turbo Debugger and Turbo Profiler. The audience paled when Eugene told us that the C compiler in question was Watcom C! Then, wonder of wonders, he introduced Ian McPhee, the president of Watcom who came to the podium to exhort the virtues of Watcom C 8.0.

That peculiar chain of events left the entire conference wondering if Borland, vendor of the respected Turbo C, had lost their corporate marbles. Does Gimbel's tout Macy's? It took me a couple of days to run down an "informed source" in Borland to get an explanation. David Intersimone, Borland's traveling evangelist, assured me that Borland is not surrendering the high-end C compiler market to Watcom or anyone else. It's just that they want to position the debugger and profiler as everyone's favorite tool collection, regardless of the compiler they use. Borland revealed the formats that a compiler must emit in order to be compatible, and invited all the other compiler folks to sign up. But so far Watcom is the only outfit to get on board. That's good news for professional development shops. You can use Turbo C and the Tools to develop a system and Watcom C for the final compile. Watcom's optimizer is one of the best.

What's New?

I saw a number of neat C products but not much that was new. Most vendors showed new versions of existing products. I was particularly impressed with a package called "Pro-C" that is trying to replace most of us. You interactively describe a system that uses screens and a data base and Pro-C from Vestronix (another Canadian developer) emits C code that you can tweak and compile to implement the system, reminiscent of the application generators of yore that crank out Cobol. I haven't tried it, but the demonstration was impressive.

The Comparative Language Panel

Warren Keuffel of Miller-Freeman invited me to represent C and C++ on this year's Comparative Language Panel at SD '90. The panel members each wrote a program to a specification prescribed by Warren, then used their programs to emphasize the strengths of their languages. Here's the specification.

"You are bidding for a lucrative contract with MegaBucks Corporation, developing all of their software. As part of the bidding process, MegaBucks has asked you to write a small program which demonstrates the strengths of your tools. The program, a gas mileage checker for the president of MegaBucks, accepts as (minimum) input the date, the number of gallons of gas purchased, and the price paid for the gas. Output is any meaningful graphical display or printed report which will impress the MegaBucks executives."

One of the speakers at SD '90 was Ken Orr who wrote and published a book in 1981 called *Structured Requirements Definition.* The book discusses the theory of output-oriented design, which says, "determine the outputs and work backwards." Warren's specification for the panel carries that idea to a new level. This is my first design that starts with any meaningful output that will impress the client.

Warren's purpose, of course, was to give us enough leeway to use our languages to their best advantage. I decided to forego fancy graphics outputs because such libraries, while available to C and C++ programmers are not intrinsic parts of the languages themselves. Instead, I used the simple standard input and output functions from

```
1987 Corvette License # JXA283

  Date    miles  gas     cost     mpg  cost/mile
-------- ----- ----- -------- ----- ---------
 2-01-90   550    40    45.00    13      0.08
 2-09-90   450    30    32.50    15      0.07
totals    1000    70    77.50    14      0.08

1989 Dodge License # HMS001

  Date    miles  gas     cost     mpg  cost/mile
-------- ----- ----- -------- ----- ---------
 2-02-90   250    10    11.25    25      0.05
 2-04-90    10     0     0.00     0      0.00
 2-04-90   122     7     7.00    17      0.06
totals     382    17    18.25    22      0.05
```

Figure 1: *Gas mileage report*

both languages and wrote filter programs that read an input file and write a report.

We each had 10 minutes for our presentations. Warren sat down front and glared between us and his watch. I felt like the guy who talks fast in the TV commercials. Even though I was doing two languages, Warren wouldn't give me 20 minutes. Must be because he works for that other magazine.

Figure 1 shows the report. The trip report displays trips by automobiles showing the date, miles traveled, gasoline purchased, cost, miles per gallon, and cost per mile. Figure 2 is the input. I assumed that the system had a data entry, validation, and sorting process, and that the input would be correct and in sequence, so I built the test input file as shown in Figure 2. The first three lines are the first car description with the license number, model year, and make. Then come five-line trip records with date, odometer in, odome-

```
JXA283
1987
Corvette

02/01/90
10550
10000
40
45.00

02/09/90
11000
10550
30
32.50

end

HMS001
1989
Dodge

02/02/90
5450
5200
10
11.25

02/04/90
5460
5450
0
0

02/04/90
5582
5460
7
7

end

end
```

Figure 2: *Report input*

(continued from page 130)
ter out, gallons bought, and money spent. The trip records for a vehicle are terminated with an "end" record. The file is likewise terminated.

At this point I made what may be an accidental dubious historic decision. I decided to write the C++ program first. After completing the C++ program, I ported it to C. This might be the only time anyone has done that. I did it because I was too lazy to start from scratch again. All the pieces were in place in the C++ program, and all I needed to do was move them around.

This reverse exercise in inertia gave me the opportunity to observe the extent to which the C++ treatment, with its almost object-oriented flavor, would influence the appearance of the C program. Would the approach to C look different than it would have if I had started from scratch? The conclusions I drew are two-edged. The program does not look substantially different than other C programs of similar scope. The difference is in how I think about the program. I designed the C++ program by beginning with the data objects and building classes. As I thought about the algorithms that a class needed, I considered them to be methods of the class, functions that are bound to the class and not executed in any other context. The C port has those same algorithms but they are not bound to the data structures, at least not in the code. They are, however, bound together in my mind. That program is object-oriented in spirit if not in substance, but you wouldn't know it by looking at it. Strange.

Let's look at the C program first. Listing One, page 146, is auto.h, the header file that describes the data structures. You can see the C++ influence right away. All the data items have object names even if only by way of a typedef. Listing Two, page 146, is auto.c, the rest of the program. This is an unremarkable program, one that reads the input and writes the report. See if you can find much in the way of C++ influence here.

Now consider the C++ system. Listing Three, page 146, is auto.hpp, a header file that describes the classes for the program. The first notable difference between this file and auto.h is that the *date* structure has a *display* function built into it. That same function exists in the C program, but it is not bound as tightly to the structure it supports. In C you could call it, pass it anything that looked like a date, and it would display something. In C++ the only way to call it is through an instance of the *date* structure.

The Automobile and Trip_Record classes have the same data members that their equivalent C structures have, but they also have constructor, destructor, and other member functions. There are overloaded functions and operators and other of the C++ extensions. Listing Four, page 148, is auto.cpp, the code for the member functions. Then Listing Five, page 148, is autorpt.cpp, the code for the application.

The distribution of code is different in the C++ program. Much of what you find in the auto.c part of the C program goes into the class member functions and could be tucked away as part of a reusable class library for the programmers who work for the purveyors of the Automobile Use Tracking Output System (AUTOS. See how easy acronyms are?)

There are two questionable shortcuts in the C++ program. The memory for the instances of Automobiles and Trip_Records comes from the free store and never gets sent back. Their allocation occurs in functions that return the objects themselves rather than the pointers, so when they are ready to go out of scope, the pointer values are not around to be deleted. This apparent insanity came as the result of some as yet unresolved conflict between me and the Zortech compiler. The problem was

(continued from page 132)
probably mine, and the deadline for the panel approached, so I took the easy way out.

Had I not had that problem I might have needed to code an overloaded assignment function for the Automobile class. Its destructor deletes the strings that contain the license number and manufacturer. If you use objects of this class as designed in an assignment, the sending object's pointers will

I decided to write the C++ program first

get deleted twice when the two objects go out of scope. A well-designed class will include an assignment function that correctly manages free store pointers in the sending and receiving objects. But because the Automobile class never gets assigned in such a way, the problem does not occur, and I did not spend the time to build an assignment function.

The point of the presentation was to stress the strengths of the languages. The strengths of C are well known. Its greatest strength is its newest, however, and that is that the standard has been approved. C is a standard language. It delivers tight, concise code, is known by many programmers, and holds the confidence of many managers. Those strengths add up to lots of jobs for programmers and almost lots of programmers to fill the jobs. C is especially suited for systems programming because of its tight code, because you can get close to the machine, and because it is loosely typed.

Don't laugh. One of the strengths of C++ is that it is strongly typed. But C++ serves different purposes with its object-oriented approach to software design and development and its facility that allows a programmer to extend the language with new data types. My favorite strength of C++ is that when the OOPS way doesn't seem to fit, I can lapse into good old reliable C.

Both languages share the advantage that when you choose them, you do not automatically sign up to a particular user interface or data base model. They are implemented on many systems with separate function and class libraries that support these things.

Other languages represented on the panel were SCHEME, Rexx, Smalltalk, and MicroStep. When the panel was over, Warren asked each of us to choose the language we would use if we needed to switch to one of the others. I had never used any of the other languages and so I chose the one I could read without already knowing it. That one was Rexx, an interpreted prototyping language.

Crotchet of the Month: Installation Programs

It was 2:00 a.m. My eyes were drooping and my jaws were locked. I hate installation programs that don't work. Zortech C++ 2.0 is a case in point. It arrived in the afternoon, and I eagerly set about to install it, anxious to use it for the exercises in a book I'm writing called *Teach Yourself C++* and for the SD '90 Comparative Language Panel just mentioned. Because the book gets into some of the features of C++ 2.0, I need a 2.0 compiler to test the exercises. Just now, Zortech is the only 2.0 compiler I have.

The Zortech installation program started out bad and went downhill after that. Its first message is one about screen refreshes. That message is not the one specified in the Installation Guide as the first one. That apparent contradiction set me to tearing through the manual trying to figure where I was or if I was running the wrong program. Then, although Zortech sent me 3.5-inch diskettes, the program tells me that it is installing from 5.25-inch diskettes. That made me worry. Next, a screen that asks which disk drive I am installing on accepts my D and immediately disappears with no reassurance that it knows where it is installing. The feeling it gave was that an error had occurred, and the doubt caused me to restart several times. When at last I figured that the program must know what it was doing, it took off, asking for the compiler disks, one by one. After copying files from Compiler Diskette #4, the program asked for Compiler Diskette #5. There is no Compiler Diskette #5. The only way out of the program was to press Ctrl-Break. The installation program nicely terminated, informing me that it had done so at my request and then locked up the system. Arrgh!

Hoping beyond hope, I assumed that the compiler files I needed had been copied and that I could copy the ones from the Tools, Debugger, and Library Source diskettes. The documentation said I could. However, it turns out that the Tools diskette is not a Tools diskette at all but a copy of the Debugger diskette. I have no tools, it seems. Well, that's OK, I'll get them later. With everything else that I thought I needed

in place, I loaded the ZTCHELP program. This is Zortech's on-line TSR help program after the fashion of similar programs available with Borland and Microsoft language products. It tells me, however, that it cannot find its data file, ZTC.HLP. The documentation says there should be one. I found a file named ZTCP.HLP and renamed it. Great, now ZTC can find its data base. I loaded the ZED editor, typed "printf" and pressed Shift-F1, the ZTCHELP hot key. One pretty window with much unreadable stuff in it. Stuff happens.

It's 11:00 the following morning. My eyes are wide open and my jaws are relaxed. A call to Zortech's technical support person reveals that a foul-up in the London office was the culprit and that it had been caught and corrected. He told me how to get the help system working and that my compiler is complete. They are sending me new diskettes complete with tools, and I am underway once again.

To the folks at Zortech and anyone else who builds tools for programmers: When a software developer can't get something as simple as the installation program and distribution diskettes right, it makes us wonder about the quality of the software that almost gets installed. Why do you developers feel the need to get cute with pretty windows that ask a few questions and then do nothing more than make subdirectories, copy files, and mangle our CONFIG.SYS and AUTOEXEC.BAT files? It would be OK if you could get the programs to work, but you can't, obviously, and you needn't have bothered trying. That sort of installation program is nice for poor old word processor and spreadsheet users who might not know about DOS and such, but we compiler users are programmers, for Pete's sake. Tell us what and where the files are and where to put them. And let us tell you where to put your stupid installation programs. Finally, try to get the distribution package right. Try to get all the files on the disks where you say they are. Try to get a little control on your quality.

ANSI Standard C

The ANSI Board of Standards Review approved the draft proposed standard for C as an ANSI standard in the December – January time frame. This is a monumental achievement that reflects years of hard work by a small number of dedicated souls called collectively the "X3J11 committee." The next several years will involve the committee in the task of interpreting the standard, which involves telling anyone who asks what this or that paragraph in the docu-

ment really means. I wouldn't want that job.

By the time this column reaches you, the official document should be available. Its name is the "American National Standard for Programming Language C." It is formally known as X3.159-1989. It is hard to read. See what follows for two reasonable alternatives.

Paperback Writer

Don't think all those dedicated X3J11 volunteers will go uncompensated. Some of them are writing books, an activity of which I heartily approve.

The small paperback book format is popular now among publishers of computer books. Every subject imaginable is out in a "quick reference guide," which means a small book and, you hope, a small price. Such a book is *Standard C* by P.J. Plauger and Jim Brodie (Microsoft Press, 1989). Both authors are officers on the X3J11 committee and, as such, can be considered authorities on the subject. This book is an excellent reference manual on the syntax and function libraries of standard C. Typical of the publishing industry, the book's publication was scheduled to coincide with the approval of the standard. The approval was delayed but the book came out anyway. No matter, you'd be hard put to find any discrepancies.

I use this book as my primary reference work when I want to write a standard C program, and it hasn't let me down yet. I have two complaints, though. The book uses the "railroad track" diagram format to describe the rules of syntax. This is a format that is generally unfamiliar to most programmers unless they design languages or write compilers. The diagrams can become complex and generally unattractive, and you will find yourself retreating to the description of the diagram format so you can understand the diagrams.

My other complaint is one of format. I'd spend less time looking things up in the book if the function descriptions were organized alphabetically by function rather than by function within the name of the header file that contains the function's prototype.

Complaints aside, I can recommend this book. Every C programmer should have it.

Mutt and Jeff

The little *Standard C* book just discussed is a pure reference work. No tutorial. If you don't already know something about C, look elsewhere. Here is where to look. Another prominent member of the X3J11 committee is Rex Jaeschke, well-known C author, consultant, and teacher. He has written a big book called *Mastering Standard C*, (Professional Press, 1989). I say big book, because instead of the little paperback format, Rex has chosen the 8.5 × 11, spiral bound format for this comprehensive tutorial on standard C. Start with this one. It uses the tutorial approach, describing the subjects in easily read English and in a sequence designed to teach rather than describe. Besides, it lays flat on your desk, staying open to the page you've selected.

Of course I have a complaint. I've been looking in vain for a use for the preprocessor's ## token pasting operator. X3J11 added it to the language, and whoever dreamed it up must have had a use for it. However, none of the written matter I've seen tells me what the heck it is for. You would not expect a reference work to tell you how you might want to use some feature, but a tutorial should offer at least a glimmer, especially when the subject is as abstruse as this token pasting operator. Rex apparently doesn't understand it either. He not-so-deftly sidesteps the issue by saying that it is too advanced to be discussed.

If you want to learn the C language and are willing to spend some time learning it end-to-end, *Mastering Standard C* is a good way to go.

"Just Say Si"

Soon we'll stop saying "ANSI C" and start saying "Standard C." Then, when the old non-conforming compilers bite the dust, we'll drop the "standard." Maybe someday, when the two finally converge, we'll stop saying "C++" and just say C.

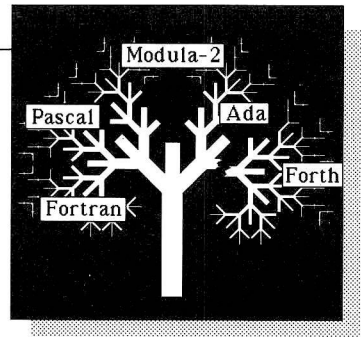
Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 146.)

Vote for your favorite feature/article.
Circle Reader Service **No. 14.**



Grinding the Speckled Axe

Jeff Duntemann KI6RA/7

There's an old story they tell up in Wisconsin:

A farm boy found an old axe chunked into a tree stump while trampin' the deep woods. He brought it home, touched up the edge with his trusty whetstone, and found that it still cut wood as well as an axe oughter. The axe head had gotten mighty rusty over the years, though, and was all pitted and rough.

So the boy asked his paw if he'd grind the axe head full bright agin on the big grinding wheel out aside the barn. And his paw said, "Sure, son — if you turn the wheel."

So the boy pushed the pedal of the big grinding wheel while his paw pressed the old axehead agin it. The sparks flew, because his paw was a strong man and held the axe head hard agin the wheel. The rust was ground away, but under it all the deepest rust spots still showed as speckles all over the new bright steel.

So the man kept the axe head pressed hard agin the wheel, knowing his son wanted it ground full-bright. But after half an hour, the speckles were fewer but still there, and the boy let up on the pedal so that the wheel stopped.

"Come on, son," his paw said, "we'll make those speckles go away afore sundown!"

But the boy, when he stopped panting, lowered his eyes and said, "Y'know, paw, I heard at school last week that speckled axes was fersure the latest thing."

The Axe or the Pits?

The next time you find yourself working long past sundown grinding the pits from a speckled axe, think hard about the very serious question: What am I working on? The axe or the pits?

It's a funny thing about this programming stuff. Unlike a lot of work I can think of (like being a soda jerk at Walgreen's or going around the backyard in March picking up what the dog's been doing all winter) it's fun. It can become so much fun that watching the sparks and pumping the pedal take on a fascination all their own, regardless of the work actually being done.

Complicating the issue is the fact that a lot of us (and I'm certainly right in there) grind away at the pits because it feels good, without needing to get anything in particular accomplished. This is why there are Mandelbrot Set programs all over the place; man, you want sparks?

Now and then I think it's well worth drawing a sharp line between work that has to be done because the job needs doing, and work that gets done because it feels good. If you get a kick out of spinning elaborate menuing and windowing systems out of thin air, the tendency is to launch in and do it all over again whenever you pick up a consulting project or get a new assignment from the boss. Remember: They're not paying you for the shine. They're paying you for the edge.

Unearthing a Truly Ugly Truth

This bears on the theme I began last month, which is the art of designing with object-oriented techniques. OOP brings a level of modularity and loose coupling to structured programming that has been almost unknown until today. The same well-designed object hierarchy can be used as the foundation for many different applications, in part because of that modularity, and in part because OOP is unexcelled at generalizing functionality. It's the ultimate in efficient boilerplating schemes: Take a generic field object and add to it only what it needs to become a date field object, or a phone number field object, or a Boolean field object. Take what you can. Add only what you must.

So what I'm going to tell you to do early-on in an OOP development project is this: Take a walk in the woods. See what's lying around, and see what you can bring back with you. Resist, as much as possible, the Not Invented Here syndrome. If it works, and if it's even remotely close to what you need, grab it. If necessary, bend your spec to align it with the architecture of the object libraries you find.

I say this for a very important reason. With going on a year of object-oriented exploration behind me, I have identified (and others have verified) a Truly Ugly Truth: Designing a workable and efficient object hierarchy is murder. Much of the ease and the magic in Smalltalk stems from the fact that its standard class libraries are works of brilliance, they're done, they have some history (Smalltalk was in "design mode" for about 15 years before becoming available for small machines), and they're yours with the language. Much the same is true of Actor, and if it doesn't have as many years of history, that's mostly because its authors built on the better features of well-known and thoroughly shaken-out languages such as C and Pascal.

This sounds a little like I'm growing disenchanted with OOP in general. Not true — but as I said in an earlier column, the hype has gotten way out of hand, especially when we don't really know what OOP's true value to the programming art will turn out to be. People have been attributing "easy, hassle-free prototyping" to the OOP nature of languages such as Smalltalk, when in fact the ease of prototyping was due to the brilliance and completeness of the class libraries. Prototyping in "naked" (that is, class-less) Object Pascal is no easier than prototyping in Ex-

tended Pascal, or whatever people will eventually come to call the Turbo Pascal standard language definition.

So if you're launching into an ambitious design project in Object Pascal or (God help us) C++, do not expect the OOP nature of the underlying language to make things easier on you. On the contrary, you will soon be up to your nether parts in some of the most intractable decisions of your entire career.

Drawing the Line Between Parent and Child

Let me give you a very simple f'rinstance, one that came up during the terrific Get-TUG-Gether seminar and party in Seattle last summer. (I'd love to shake your hand up there this year — information is in the "Products Mentioned" box at the end of this column.) I was explaining polymorphism to the group, and using the example of a field editor class. I drew a figure on the whiteboard that looked something like the one shown in Figure 1.

The idea was to create an abstract class named *Field*, which would never actually be instantiated. *Field* would take care of all those things that every field — regardless of its type — would have to do: Clear some number of characters in a line, lay down a title, indicate that this particular field is active, and so on. But *Field* would not actually have any "guts" with which to edit data. Instead, child objects of *Field* would be derived from *Field* to do the actual editing, through an *Edit* virtual method present in each.

In Figure 1, I show them as *StringField*, *IntegerField*, and *DateField*. A *StringField* object would simply accept string data and allow simple line-type edits on the string data. An *IntegerField* would allow entry of only numeric digits and supporting characters like the minus sign. If the user attempted to press "W" while executing *IntegerField.Edit*, the object would beep and refuse the character. Similarly, *DateField.Edit* would validate a date character by character, perhaps within a template separating day, month, and year subfields. In any event, you edit the data in any of these related objects simply by executing an object's *Edit* method.

This works — I've actually written the code since then — but as the estimable Brian Foley of Turbo Power Soft-

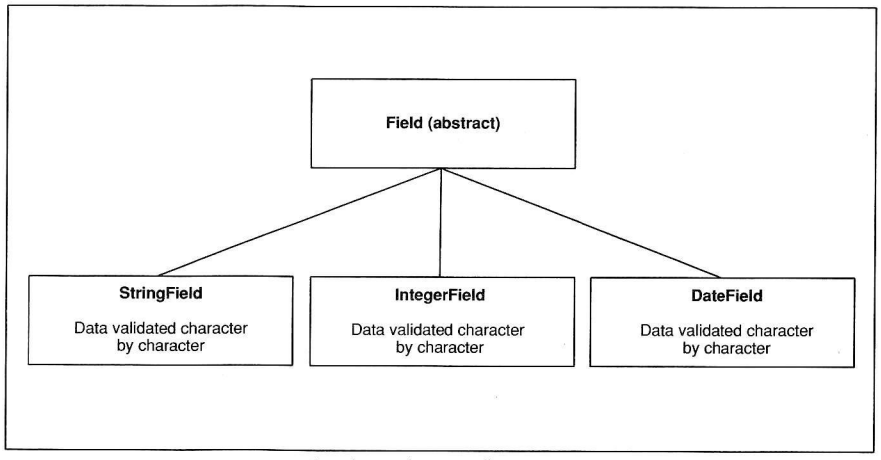


Figure 1: *First try at a simple object hierarchy*

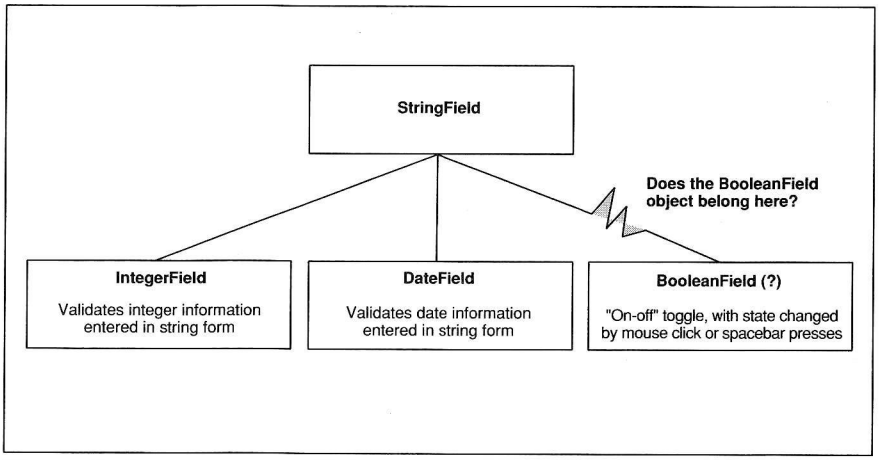


Figure 2: *A more memory-efficient arrangement*

ware rose to point out, this system is very inefficient from a code standpoint. The three child objects shown in Figure 1 have a lot of code in common, because all three control a cursor within a delimited portion of a line, picking up characters from the keyboard and formatting them on the screen.

Brian took the time to rearrange the figure much as I've shown it in Figure 2. He placed virtually all of the field editing code in a parent object named *StringField*, because beneath it all, we encode most data in string form anyway. What actually differs among the various types of data is how the data is validated. The child objects all gather information from the user by calling their collective parent's *Edit* method, and then examine that data for adherence to the specific requirements of the data type after the user has pressed Enter. The *StringField* parent class has no validation, because any string data enterable by the user is legal tender. *StringField* is, however, a usable, nonabstract class.

Brian's arrangement eliminates nearly all duplication of code among the various objects. And like my earlier arrangement, it also works. The line is drawn differently between parent and child. Which is best?

There's no single answer. And it's an even messier question than I've shown so far. Suppose I want to add a *BooleanField* object to the hierarchy? That is, I want a field to select between two available choices like Yes/No, Male/Female, Active/Suspended, Alive/Deceased. But I don't want the user to have to actually type out either of the two values. I want each field to have a default value, and to allow the user to toggle between the two values for a given field with the click of a mouse or a press on the space bar.

It gets pretty obvious that *BooleanField* needs to be descended from something else, something probably one level higher than *StringField*. Also, entering the full value of a field in string form and then validating the string after Enter is pressed forbids something I like a lot: Character-by-character entry validation. In Figure 1, an *IntegerField* will beep a "W" the moment it's pressed. In Figure 2, the "W" won't be detected or an error reported until the user presses Enter. The user has to go back and "fix" the data — when initially, he or she wouldn't have been allowed to enter invalid data to begin with.

You'll find that subtle changes in the structure of an object hierarchy have lots of odd little side-effects like that. You might leap initially to Brian's arrangement in a quest for code efficiency, only to find that user operability of the application suffers.

Distribution or Extension

I've run into this little conundrum a time or two since then. There seem to be two general ways to construct an object hierarchy. The line between the two is not sharply drawn, and in fact it may not be a line at all — it's less a difference of technique than a difference of emphasis. My first approach emphasized the distribution of functionality through an abstract class. When you take this approach, you often design a whole tree full of relatively small and simple abstract classes, none of which are capable of instantiation. Each abstract class adds something fairly simple, conceptually, to the hierarchy. From these abstract classes you derive the "leaves" of the hierarchy tree, which are the fully functional classes that may be instantiated and do real work.

The other approach, by contrast, emphasizes fully workable objects almost from the start. The first objects designed lean toward the general-purpose — as in Brian's string editor — but they are nonetheless fully functional objects. Abstract classes are kept to a minimum, and the more general objects are extended to produce additional objects, like the *IntegerField* and *DateField* objects.

The two approaches differ in another fundamental way. Using many abstract classes allows you to cast a great many objects into just two or three major object hierarchies. Avoiding abstract classes tends to produce numerous small object hierarchies, sometimes with only two or three classes in each.

OK. Which approach is better? Hear me well: We don't know. This is not the same as saying, It doesn't matter. There is no information as to whether maintainability or reusability is affected over time by the approach taken to OOP design. For every question you unearth, the answer only yields two or three more questions.

Getting Bananas the Hard Way

Using one or two monster hierarchies has the downside of giving you the whole gorilla even when you only want the banana, as so aptly put by Scott Guthery in his article in the December 1989 *DDJ*, "Are the Emperor's New Clothes Object Oriented?" My answer to Scott is fairly simple: Hey, guy, anybody who chases a gorilla to get a banana deserves whatever he gets. I get my bananas from the produce department at Safeway. Gorillas are not involved in the transaction.

And that's a point that I'm not sure has been made as clearly as it should: OOP is a tool that can be brought to bear on different problems, not all of them requiring a massive hierarchy and pervasive polymorphism. OOP can be used to enforce modularity on small structures, such as the "when" stamp I discussed in last month's column. The when stamp is a banana; it stands pretty much alone and can be used and reused as a modular unit.

I suspect the smart path is to build large object hierarchies only when the hierarchy as a whole is the mechanism you're after. In other words, don't glue together diverse and unrelated items into a single hierarchy just because it seems the politically correct thing to do. I have seen an unannounced product that is, essentially, the infrastructure of an event-driven application, rolled up into a single large and brilliantly conceived object hierarchy. It incorporates text windowing, mouse and keyboard input, dialog boxes, various types of buttons and controls, an event processing loop, and so on — everything but the parts of the app that do the specific work the application was designed to do. Trying to find and extract a "window" object in the hierarchy for reuse is nonsensical —the window-ness of the objects in the hierarchy is distributed in a highly stochastic fashion across the hierarchy. If you want the infrastructure, you take it all or leave it on the shelf.

After all, trying to extract the liver from a gorilla will at best leave you with something not especially useful — and it annoys the hell out of the gorilla.

Object Professional Arrives!

By the time you read this, Turbo Power Software's massive Object Professional library will be on the streets. I've been watching the product grow for some time, and I have beside me a galley proof of the documentation. It's quite a product, and I'll be having more to say about it in the future.

Anybody who chases a gorilla to get a banana deserves whatever he gets

OPRO, as the folks at Turbo Power have begun to call it informally, is an evolutionary step up from Turbo Professional 5.0. It's a massive collection of about 130 object types in 50 units, with 1600 pages (!) of documentation — something that may set a new record for a toolbox product, and one likely to stand for a good long while.

OPRO follows the "banana model" of object libraries. It's a truck load of relatively small hierarchies, most of which are independent of one another. The largest aggregate of functions supports a slick text windowing system (OPRO does not involve graphics) with overlapping windows, shadows, scroll bars, pick lists, menus, and context-sensitive help windows.

I could spend a whole column just describing what's in OPRO, but space is short. Here are the highlights:

- A printing manager for your programs that includes a mechanism for creating and using printer device drivers as well as a forms and reports printing mechanism.
- A multiline text memo field object, a full-blown text editor object, and a text file browser object.
- Long-array objects that let single arrays break the 64K barrier, with data stored in DOS memory, EMS, or on disk.
- Container class objects for creating stacks, queues, linked lists, and so on.

• Swapping TSR support. This is something pretty unprecedented as far as I can tell: A mechanism by which you can create TSR programs that occupy as little as 6K of DOS RAM. The balance can be stored in EMS (but not extended memory) or on disk, which includes RAMdisks. The TSRs you write are transparently protected against DOS re-entrancy problems. I'll cover this in an upcoming column in more detail — amazing.
• A drop-in keyboard macro facility for your programs.
• A data-entry screen manager with an interactive screen forms designer.
• A swapping DOS shell.

That's not all, either, but I've not had the time to go over more than a fraction of it so far. Most remarkable of all, every last bit of source code is included, right down to the .ASM files. Just reading the code is a helluvan education in creating efficient objects in Turbo Pascal.

OPRO was not an easy product to write, even for Kim Kokkonen and his crew of wizards down on Scotts Valley Drive. (I could hear the groans drowning out the sawmill regularly when I had the window open in the fall.) This is why I recommend that when an application has to happen in the least amount of time, glom onto a major toolkit product and write your application around it. The toolkit may not have every corner of every window done just the way you want it. There are, for example, operational details about Object Professional's windowing system that I don't care for, but these are speckles on the axe. Keep reminding yourself: You're working on the edge, not the shine.

If you can buy the shine somewhere, do it. And don't assume that you can necessarily do a better job in less than a geological eon of pressing the axe against the stone.

Market Notes

As most of you who have ever considered marketing your software professionally are aware, it's virtually impossible to get a product on computer store shelves without a tremendous budget for promotion and more than a little luck. So I watch for ways that a small time operator can make some money with their software, and another good one has turned up recently.

Bob Napp, managing editor of SoftDisk Publishing, is looking for good software to publish in their monthly disk-based magazine *Big Blue Disk.* Billed as "The Monthly Software Collection," *Big Blue Disk* is an amiable hodgepodge of DOS software, running from batch language extension utilities to D&D-style games. A recent issue that I examined contained a typing tutor, a dungeon game, a video board detection utility for batch files, an appointment calendar manager, a simple word processor, and a collection of clip art for the Print Shop desktop publishing program, plus a letters column and a few other odds and ends. From the consumer's standpoint it was well worth the $9.95 cover price, and their rates for authors, while not the stuff of which empires are made, certainly seem worth it to me for modest software projects.

Contact Bob at SoftDisk Publishing and get their author kit.

Apologies

. . .to the C people who have had enough of my C bashing. Truce, truce. After all (and yes, I have violated the spirit of this contention from time to time) a language is a socket wrench, not a religion or a household pet.

And the truce will hold as long as you C guys stop asking me when I'm going to give up my training wheels.

Deal?

Contact Jeff Duntemann on MCI Mail as JDuntemann, or on CompuServe as 76117,1426.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 15.**

Products Mentioned

Big Blue Disk
SoftDisk Publishing
P.O. Box 30008
Shreveport, LA 71130-0008
$9.95 per issue

Get-TUG-Gether
June 29 – July 1 in
Silverdale, Washington
Conference fee $95
Turbo User Group
P.O. Box 1510
Poulsbo, WA 98370
206-779-9508

Object Professional 1.0
Turbo Power Software
P.O. Box 66747
Scotts Valley, CA 95066
408-438-8608
$150 (includes full source code)

Listing One *(Text begins on page 129.)*

```
/* --------- auto.h ------------ */

/*
 * simple data element classes
 */
typedef long odometer;
typedef double Dollars;

/*
 * a simple date class
 */
typedef struct {
    int mo, da, yr;
} date;

/*
 * Automobile class
 */
typedef struct {
    int modelyear;
    char manufacturer[20];
    char license[9];
} Automobile;

/*
 * Trip_Record class
 */
typedef struct {
    date trip_date;
    odometer odometer_in;
    odometer odometer_out;
    int gallons;
    Dollars cost;
} Trip_Record;
```

End Listing One

Listing Two

```
/* --------- auto.c ------------ */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "auto.h"

static Automobile get_car(void);
static void reportauto(Automobile);
char *display_date(date dt);
static Trip_Record get_trip(void);
static void reporttrip(Trip_Record);

void main()
{
    while (1)   {
        /* ---- a report for each car in the input ----- */
        Automobile car = get_car();
        Trip_Record trip_totals = {{0,0,0},0,0,0,0};

        if (car.modelyear == 0)
            break;
        reportauto(car);
        /* -------- prepare the detail report ---------- */
        printf("\n  Date   miles  gas    cost    mpg  cost/mile");
        printf("\n-------- ----- ----- -------- ----- ---------");
        while (1)   {
            /* ----- get the next Trip_Record from the input ---- */
            Trip_Record trip = get_trip();
            if (trip.trip_date.mo == 0)
                break;
            /* -------- report the trip ----------- */
            reporttrip(trip);
            /* -------- collect totals for this car -------- */
            trip_totals.odometer_in =
                max(trip_totals.odometer_in, trip.odometer_in);
            trip_totals.odometer_out = trip_totals.odometer_out ?
                min(trip_totals.odometer_out, trip.odometer_out) :
                trip.odometer_out;
            trip_totals.gallons += trip.gallons;
            trip_totals.cost += trip.cost;
        }
        reporttrip(trip_totals);
    }
}

/* ---------- get a car's description from the input -------- */
static Automobile get_car(void)
{
    static Automobile car;

    car.modelyear = 0;
    scanf("%s", car.license);
    if (strcmp(car.license, "end") != 0)    {
        scanf("%d", &car.modelyear);
        scanf("%s", car.manufacturer);
    }
    return car;
}
 static void reportauto(Automobile car)
 {
     printf("\n\n%d %s License # %s\n",
         car.modelyear, car.manufacturer, car.license);
 }

 /* ------ get a trip record from the input stream ----- */
 static Trip_Record get_trip(void)
 {
     date dt;
     char strdate[9];
     odometer oin = 0, oout = 0;
     int gas = 0;
     Dollars cost = 0;
     static Trip_Record trip;

     dt.mo = 0;
     scanf("%s", strdate);
     if (strcmp(strdate, "end")) {
         dt.mo = atoi(strdate);
         dt.da = atoi(strdate + 3);
         dt.yr = atoi(strdate + 6);
         scanf("%ld", &oin);
         scanf("%ld", &oout);
         scanf("%d", &gas);
         scanf("%lf", &cost);
     }
     trip.trip_date = dt;
     trip.odometer_in = oin;
     trip.odometer_out = oout;
     trip.gallons = gas;
     trip.cost = cost;
     return trip;
}

static void reporttrip(Trip_Record trip)
{
    int miles = (int) (trip.odometer_in - trip.odometer_out);
    printf("\n%s %5d %5d %#8.2f %5d %#8.2f",
                trip.trip_date.mo ?
                    display_date(trip.trip_date) :
                    "totals  ",
                miles,
                trip.gallons,
                trip.cost,
                trip.gallons ? miles / trip.gallons : 0,
                miles ? trip.cost / miles : 0
                );
}

char *display_date(date dt)
{
    static char d[9];
    sprintf(d, "%2d-%02d-%02d", dt.mo, dt.da, dt.yr);
    return d;
}
```

End Listing Two

Listing Three

```
// auto.hpp

// ------------------------------
// simple data element classes
// ------------------------------
typedef long odometer;
typedef double Dollars;

//  ------------------------------
// a simple date class
// ------------------------------
struct date {
    int mo, da, yr;
    char *display(void);
};

// ------------------------------
// Automobile class
// ------------------------------
class Automobile    {
private:
    int modelyear;
    char *manufacturer;
    char *license;
public:
    Automobile(char *license, int year, char *make);
    Automobile();
    ~Automobile();
    char *display(void);
    int nullcar(void) {return modelyear == 0;}
};

// ------------------------------
// Trip_Record class
// ------------------------------
class Trip_Record   {
private:
    date trip_date;
    odometer odometer_in;
    odometer odometer_out;
    int gallons;
    Dollars cost;
public:
    Trip_Record(void);
    Trip_Record(date, odometer, odometer, int, Dollars);
    int mileage(void) {return odometer_in - odometer_out;}
    char *display(void);
```

(Listing continued on page 148)

Listing Three *(Listing continued, text begins on page 129.)*

```
    int nulltrip(void) {return trip_date.mo == 0;}
    void operator += (Trip_Record&);
};
```

End Listing Three

Listing Four

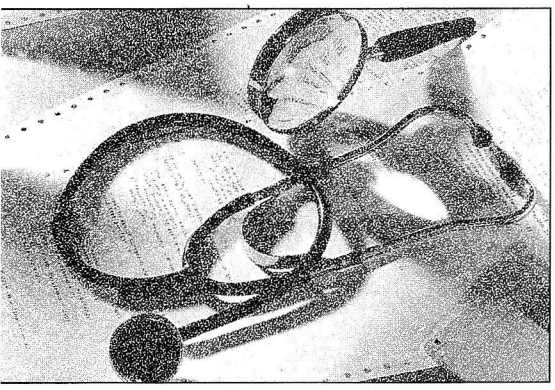
```
// auto.cpp

#include <stream.hpp>
#include <string.h>
#include <stdlib.h>
#include "auto.hpp"

char *date::display()
{
    return form("%2d-%02d-%02d", mo, da, yr);
}

// --------------- constructors ---------------------------
Automobile::Automobile()
{
    modelyear = 0;
    manufacturer = license = NULL;
}

Automobile::Automobile(char *lic, int year, char *make)
{
    license = new char[strlen(lic)+1];
    strcpy(license, lic);
    modelyear = year;
    manufacturer = new char[strlen(make)+1];
    strcpy(manufacturer, make);
}

// --------------- destructor -------------------
Automobile::~Automobile()
{
    if (license != NULL)
        delete license;
    if (manufacturer != NULL)
        delete manufacturer;
}

char *Automobile::display()
{
    return form("%d %s License # %s",
                modelyear, manufacturer, license);
}

#define max(a,b) ((a)>(b)?(a):(b))
#define min(a,b) ((a)<(b)?(a):(b))

void Trip_Record::operator += (Trip_Record& trip)
{
    this->odometer_in = max(this->odometer_in, trip.odometer_in);
    this->odometer_out = this->odometer_out ?
                         min(this->odometer_out, trip.odometer_out) :
                         trip.odometer_out;
    this->gallons += trip.gallons;
    this->cost += trip.cost;
}

// --------------- constructors ---------------------------
Trip_Record::Trip_Record()
{
    trip_date.mo = trip_date.da = trip_date.yr = 0;
    odometer_in = odometer_out = 0;
    gallons = 0;
    cost = 0;
}

Trip_Record::Trip_Record(date trdt, odometer oin, odometer oout,
                         int gas, Dollars cst)
{
    trip_date = trdt;
    odometer_in = oin;
    odometer_out = oout;
    gallons = gas;
    cost = cst;
}

char *Trip_Record::display()
{
    int miles = mileage();
    return form("%s %5d %5d %#8.2f %5d %#8.2f",
                trip_date.mo ? trip_date.display() : "totals  ",
                miles,
                gallons,
                cost,
                gallons ? miles / gallons : 0,
                miles ? cost / miles : 0
                );
}
```

End Listing Four

Listing Five

```
// autorpt.cpp

#include <stream.hpp>
#include <string.h>
#include <stdlib.h>
#include "auto.hpp"

// ========================================================
// automobile operating costs system
// ========================================================

// ------- prototypes
static Automobile& get_car(void);
static Trip_Record& get_trip(void);
static void report(Automobile&);
static void report(Trip_Record&);

main()
{
    while (1)   {
        // -------- a report for each car in the input
        Automobile car = get_car();
        if (car.nullcar())
            break;
        report(car);
        // -------- prepare the detail report
        cout << "\n  Date   miles  gas    cost    mpg  cost/mile";
        cout << "\n-------- ----- ----- -------- ----- ---------";
        // ---------- a Trip_Record to hold the totals
        Trip_Record trip_totals;

        while (1)   {
            // -------- get the next Trip_Record from the input
            Trip_Record trip = get_trip();
            if (trip.nulltrip())
                break;
            // -------- report the trip
            report(trip);
            // -------- collect totals for this car
            trip_totals += trip;
        }
        report(trip_totals);
    }
}

// ---------- get a car's description from the input
static Automobile& get_car(void)
{
    char license[7];
    int modelyear = 0;
    char make[25] = "";

    cin >> license;
    if (strcmp(license, "end") != 0)    {
        cin >> modelyear;
        cin >> make;
    }
    Automobile *car = new Automobile(license, modelyear, make);
    return *car;
}

static void report(Automobile& car)
{
    cout << "\n\n" << car.display() << '\n';
}

// ---------- get a trip record from the input stream
static Trip_Record& get_trip()
{
    date dt;
    dt.mo = 0;
    char strdate[9];
    odometer oin = 0, oout = 0;
    int gas = 0;
    Dollars cost = 0;

    cin >> strdate;
    if (strcmp(strdate, "end")) {
        dt.mo = atoi(strdate);
        dt.da = atoi(strdate + 3);
        dt.yr = atoi(strdate + 6);
        cin >> oin;
        cin >> oout;
        cin >> gas;
        cin >> cost;
    }
    Trip_Record *trip = new Trip_Record(dt,oin, oout, gas, cost);
    return *trip;
}

static void report(Trip_Record& trip)
{
    cout << '\n' << trip.display();
}
```

End Listings



Encapsulating C++ Books

C++ resources for your bookshelf

Andrew Schulman

For a long time, if you wanted to learn C++ and happened not to work down the hall from its inventor, Bjarne Stroustrup, the only resource you could turn to was Stroustrup's book, *The C++ Programming Language* (Reading, Mass.: Addison-Wesley, 1986, 328 pages, $22.95). But trying to learn C++ from the Stroustrup book is like trying to learn chemistry from the Periodic Table of the Elements. That situation has changed, however, and there are finally some good books from which you can learn C++.

The definitive book on C++ 2.0 is Stanley B. Lippman's *C++ Primer* (Reading, Mass.: Addison-Wesley, 1989, 464 pages, $31.25). Like Stroustrup, Lippman works at AT&T Bell Laboratories. This is a tutorial, not a reference. After a brief discussion of the more C-like features of C++, Lippman moves quickly into a discussion of class. The book presents thorough discussions of the three key features of object-oriented programming: abstract data types, inheritance, and dynamic binding. There are also several discussions of when not to use such C++ features such as operator- and function-overloading. I might quibble with a few aspects of the book, such as the odd interface for a BitVector class, but if you can afford only one C++ book, this is the one to get. While the book assumes the reader is using C++ 2.0 (and the unsuspecting reader using C++ 1.2 might get tripped up in a few places), an appendix presents a clear discussion of the differences between C++ 1.2 and 2.0. Another appendix discusses the differences between C and C++.

Like Stroustrup's book, though, Lippman's primer largely presents the language features, and provides little direction on how to *use* C++. Another new C++ book makes up for this deficiency by encouraging the reader to think in C++, and introduces general principles of data abstraction and object-oriented programming: *Programming in C++*, by Stephen C. Dewhurst and Kathy T. Stark (Englewood Cliffs, New Jersey: Prentice Hall, 1989, 233 pages, $22.00). They've included some cutesy code in this book (for example, a File class in which the class constructor opens the file and the destructor closes the file, and an iterator object that uses *operator( )( )*), but one of the authors' goals is to show how C++ supports a wide variety of programming methods and techniques. An entire chapter is devoted to libraries, a crucial topic entirely missing from other books on C++. This book appears in the superb "Prentice Hall Software Series," edited by Brian Kernighan.

Object-oriented design is largely bottom-up, emphasizing the creation of reusable classes that are then soldered together to form different applications. Brad Cox, author of Objective-C, has popularized this notion under the name "Software-IC" (which Cox unfortunately then went on to trademark). True to its spartan C heritage, however, C++ compilers don't come with extensive libraries of such reusable classes. Therefore, beginning C++ programmers need, above all, some useful classes such as those that Smalltalk programmers take for granted.

One new C++ book addresses this problem by providing you with tons of reusable classes: *The C++ Answer Book*, by Tony L. Hansen (Reading, Mass.: Addison-Wesley, 1990, 578 pages, $26.95). As the *C Answer Book* is to K&R, so the *C++ Answer Book* is to Stroustrup. It provides answers for every exercise in the book. For example, Exercise 6.11 is "Define a class implementing arbitrary precision arithmetic"; Hansen's answer runs over 30 pages. Exercise 7.10 (marked as difficulty-level *5 by Stroustrup) is "Design and implement a library for writing event-driven simulations. Hint: <task.h>"; Hansen's answer runs almost 40 pages. As the book is tied to Stroustrup, there is little use of C++ 2.0 here. Surprisingly, the book not only employs Unix, but also recognizes the importance of MS-DOS. In addition to using AT&T's cfront Translator, the answers were tested on the Zortech C++ compiler 1.5. Where exercise 5.5 for defining an arithmetic-expression class also suggests toying with different ways of printing

Andrew is a software engineer in Cambridge, Mass. and is a contributing editor to DDJ. *Andrew can be reached at 32 Andrew St., Cambridge, MA 02139.*

the expression, including assembly code, Hansen proceeds to do so using not only the AT&T WE32100, but also the Intel 8086. An amazing book! My only quibble is the large number of typographical errors (like most books out of AT&T, this one was typeset by the author). The forthcoming fourth printing will correct these errors.

Where does C++ fit into the hierarchy of programming languages? How does it differ from Smalltalk? from Modula-2? from Ada? from C? If you are interested in programming languages in general, and want to view C++ in some larger context, get Ravi Sethi's textbook, *Programming Languages: Concepts and Constructs* (Reading, Mass.: Addison-Wesley, 1989, 478 pages, $40.95). Sethi is one of the co-authors of the famous "dragon book" on compilers and, again, works at Bell Labs. Sethi's chapters on "Data Encapsulation" and "Inheritance" both contain extensive discussions of C++.

In general, when there are both trade and academic books on a subject such as C++, it is safer to go with the academic book. Fortunately, this is changing: Bruce Eckel's *Using C++* (Berkeley: Osborne/McGraw-Hill, 1989, 617 pp., $24.95) is a wonderful book. I don't normally buy books from Osborne/McGraw-Hill (which publishes a lot of books that are either identical to, or worse than, the manuals they purport to supplement), but I made an exception here because of Eckel's name on the cover (Bruce wrote the brilliant "Programmer's Guide to the Parallel Port" in Vol. 1, No. 1 of the late lamented *Turbo Technix*). One of the interesting examples is a program for the game of *Life*. Bruce realized that *Life* is actually a problem in simulation and modeling, so his code presents a framework in which the rules of the simulation can be easily changed. Other examples include a tiny AWK interpreter named TAWK (which appeared in *DDJ* in May 1989), a clock-based control system, and a drawing program called MicroCAD, which uses the mouse and graphics package from Zortech C++. All the code is available on disk. Great stuff!

If you are already proficient in C++ and are looking for advanced material, check out the proceedings of the C++ conferences and workshops given by USENIX (*USENIX Proceedings*, C++ Workshop, Santa Fe, New Mexico: 1987, 468 pp., and *USENIX Proceedings*, C++ Conference, Denver, Colo., 1988, 362 pp., USENIX Association, P.O. Box 2299, Berkeley, CA 94710, $30.00 each). Some trade publisher ought to pick these up and make them more widely available.

The Denver volume includes Stroustrup's papers on "Parameterized Types for C++" and "Type-safe Linkage for C++." Other papers include "Exception Handling without Language Extensions" by William M. Miller, and "Pointer to Class Members in C++" by Lippman and Stroustrup.

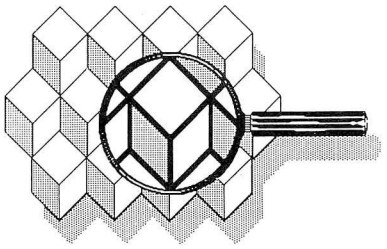
The Santa Fe proceedings includes these papers by Stroustrup: "The Evolution of C++ from 1985 to 1987," "What is 'Object-Oriented Programming'?" "Possible Directions for C++," and "A Set of C++ Classes" (with Jon Shopiro). Other useful papers in the Santa Fe volume are "Two Extensions to C++: A Dynamic Link Editor and Inner Data" by Gautron and Shopiro, "C++ on the Macintosh" by Friedenbach, and "An Object-Oriented Class Library for C++ Programs" by Keith Gorlen.

Those who disagree with pushing C++ in the direction of Smalltalk, and want to keep the language closer to its C origins, would still benefit from examining Gorlen's code

Keith Gorlen is at the National Institute of Health (NIH), and is the author of the NIH class library, a Smalltalk-like class hierarchy for C++ (class Object is at the root of the class inheritance tree). Even those who disagree with this pushing of C++ in the direction of Smalltalk, and want to keep the language closer to its C origins, would still benefit from examining Gorlen's code, which is probably the largest body of public-domain C++ source code available today. Michael Murphy of Cambridge, Mass. ported the NIH library to MS-DOS, and it may be found on the ImageSoft BBS (516-767-9074) and on BIX (listings area c.plus.plus). There is also an active C++ forum (c.plus.plus) on BIX, in which Bjarne Stroustrup (bstroustrup) participates.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 9.**



A new release of Think Pascal, Version 3.0, now includes **Symantec's** Think Class Library and support for MacApp, the class library from Apple. The Think Class Library has code for all the standard Macintosh user interface components and behaviors. This library can be used with either Think Pascal or C. And Think Pascal provides extensive support for object-oriented programming.

DDJ spoke with Claris' Dennis Cohen and Stonecutter Software's Don Sawtelle, two programmers who have been using Think Pascal 3.0 throughout its beta testing stage. Cohen said "I do all my final builds in Think Pascal 3.0. The environment is great. For an individual programmer it's phenomenal. For a team, MPW is probably better because of the projector and the tools for use with the projector." Sawtelle added that "It's [3.0] excellent. There's nothing better in terms of delivering commercial applications, because of its speed and the excellent interface for source-level debugging." Sawtelle didn't see the lack of a Projector as much of a problem: "It's so compatible with MPW, you can run both simultaneously under Multifinder."

Also new to Think Pascal 3.0 is the class browser, a graphical navigation aid that lets you view the interrelationship of your program's classes. Think Pascal 3.0 has an integrated environment, instant incremental linking, and source-level debugging tools. Productivity enhancements include better segmentation control, improved MPW compatibility, resource editing tools, and user interface enhancements. The retail price is $249, and upgrades cost $69. Reader service no. 20.
Symantec Corporation
10201 Torre Ave.
Cupertino, CA 95014-2132
408-253-9600

Version 6.0 of Microsoft C Professional Development System has been announced by **Microsoft**. Microsoft C 6.0 includes performance-oriented improvements in the compiler as well as additional software development tools, including an integrated development environment called the Programmer's Workbench.

New optimization technology added to the compiler includes register-based parameter passing, a globally optimizing code generator, segment-based pointers to memory, and pragma-level (function-level) control over optimizations. The compiler now allows in-line assembly language in the style of Quick-C.

The Programmer's Workbench is new to Microsoft C, and is an integrated development environment similar to that of Quick-C, with pull-down menus, dialog boxes, and windowing capabilities. With the Workbench are updated versions of CodeView, the make utility, the Microsoft Editor, and source browser. The Workbench's open architecture allows third-party vendors to offer additional tools.

The Source Browser is a program navigation tool that lets you view the calling structure (call tree) of an entire application, find the definition of a variable or function, or visit all references to a variable or function.

The new version of CodeView can now reside in extended memory, leaving only a 15K footprint in conventional DOS memory. CodeView now supports multiple files and multiple views into memory, browsing through structures and arrays, dynamic record-and-replay of debugging sessions, debugging of multi-threaded programs and DLLs, and remembering of configuration data from one session to another.

Improvements in OS/2 support now allow multi-threaded DLLs (by contrast, C 5.1 allows only multi-threaded apps or single-threaded DLLs). The compiler will run under either DOS or OS/2, and, under OS/2, will allow components of the compiler to operate in the background. Both the compiler and linker now have an incremental processing mode for faster operation.

The recommended configuration is 640K RAM, 10 Mbytes of hard disk space, and 384K Extended Memory or LIM 4.0. The suggested retail price is $495. Microsoft is also releasing a new version of QuickC with QuickAssembler. Reader service no. 32.
Microsoft Corporation
1 Microsoft Way
Redmond, WA 98052
206-882-8080

Insite Peripherals recently announced availability of its I325 Floptical Disk Drive to OEMs. The Floptical Drive makes use of optical technology to store up to 20 Mbytes on a standard high-density 3.5-inch diskette that embeds optical servo tracks into the surface of the media. The drive includes an optical system that detects these "grooves" and converts the images to electronic signals, which are used for servo tracking information. Magnetic data is written to "land" areas between neighboring grooves.

The company also notes that its I325VM version, which additionally reads and writes to standard 720K and 1.44-Mbyte diskettes, will be available in volume to OEMs later this year. The company claims that the Floptical's SCSI interface makes it two to three times faster than typical floppy drives and approximately 50 percent as fast as a Winchester drive. I325 evaluation kits, which include the Floptical Drive and a PC to AT host adapter, are available to OEMs for $600. Reader service no. 21.
Insite Peripherals
4433 Fortran Dr.
San Jose, CA 95134
408-946-8080

A series of modular tools for memory management is available with the utility program Above DISC, Version 3.1, from **Above Software**. Above DISC is an expanded memory simulation (EMS) program for DOS users. Some of the added features are a hard disk spill-over with disk caching for hard disk emulation, and the ability to increase conventional memory to 736K on systems with EGA/VGA adaptors.

The hard disk spill-over feature is for 286 and 386 users. It increases memory without additional RAM chips, converting extended memory to expanded memory for applications that require large amounts of expanded memory. The conventional memory enhancement on EGA and VGA color systems is for all PC/XT/AT, 286, and 386 users, and comes from accessing an additional 96K from the video memory on the system's motherboard. And you can turn this feature on and off, as when you need full access to your EGA/VGA graphics capability. Above DISC comes bundled with AboveLAN, for moving Novell NET3.COM or NET4.COM drivers into the first 64K of extended memory in order to run larger applications such as Excel off of the network. Also bundled is AboveMEM, which converts up to 512K of extended memory into EMS for increased speed with Lotus 1-2-3. Above DISC 3.1 sells for $119. Reader service no. 22.
Above Software Inc.
3 Hutton Centre
Santa Ana, CA 92707
714-545-1181

(continued on page 158)

(continued from page 153)
A productivity tool for C programmers is available from **Silico-Magnetic Intelligence**. Better-C is a generic C program methodology and generator that works with most popular C compilers. According to Kalman Toth, chief scientist on the project, the product is based on complexity management, natural language naming, top-down design, and programming with objects. One of the benefits of Better-C is its pliability for projects involving teams of programmers.

Ed Hoffman of IDG Consulting told *DDJ* that his shop used Better-C to convert one of their programs to C. "One of our products is an AI system that had been written in a whole system of languages by many different programmers. There were over 200,000 lines of good code, but it got really hard to find things, and we had duplicate functions. Better-C helped us to get organized — to recode the project. We answered a few questions in the beginning, and it assigned standard names for us. It doesn't generate modules you can execute, but it helps you write better C code — it can make you look like a very well organized C programmer."

An initial module frame generator automatically generates a framework for a new module — it sets up the necessary files and spans a framework for the new module. Objects are treated as files — they can be lists, trees, databases, windows, or whatever. The *open* function gets object handles and the *close* function destroys objects. And Better-C provides exact methods for organizing programs, by using encapsulated modules as building blocks. Better-C costs $98, and the Frame Generator Source code also costs $98. Reader service. no. 23.
Silico-Magnetic Intelligence
24 Jean Lane
Chestnut Ridge, NY 10952
914-426-2610

If embedded-systems engineering is your field, you might want to look into **Avocet's** AvCase, an integrated set of tools for the code-compile-test cycle of your software projects. According to the company, they integrated the editor, C compiler, assembler, linker, and simulator/source-level debugger. And everything supposedly runs on your PC.

One keystroke can compile, assemble, and link an application. When it finds an error, the offending module and associated error message are displayed in a window for editing. The complete package includes an AvCase 8051 family C compiler with run-time library and source, user manuals, and on-line reference database; an AvCase 8051 family Assembler with linker, object librarian, hex file utility, cross-reference utility, integrated editor, make utility, user manuals, and on-line reference database and engine; and an AvCase 8051 family Simulator with source-level debugging and user manual. This whole package costs $1895. Sold separately, the C compiler and assembler package is $995, the assembler alone is $495, and the simulator package alone is $995. Requires an IBM PC or compatible, MS-DOS 2.0 or later, 640K RAM, and a hard disk. Reader service no. 25.
Avocet
120 Union St.
P.O. Box 490
Rockport, MA 04856
207-236-9055

More memory movement is possible with Move 'Em, a program loader from **Qualitas**. Move 'Em moves device drivers, network system drivers, and memory-resident programs to the formerly inaccessible memory between 640K and 1 Mbyte. Move 'Em works on all PC platforms from the XT to the 486.

A resident program optimization facility automatically determines the best loading order and required parameters for memory-resident programs and device drivers. On systems equipped with a Chips and Technologies ChipSet, the built-in shadow RAM (about 384K of memory) can be recovered by Move 'Em.

Move 'Em runs on any IBM PC or compatible with an EMS 4.0 memory board with at least 256K of EMS memory, and PC- or MS-DOS 3.0 or later. It retails for $89.

Also from Qualitas: 386MAX and 386MAX Professional, designed for PCs that use the 80386 or 80486. These management programs locate unused pockets of addresses between 640K and 1 Mbyte and move TSRs, device drivers, and other memory-resident programs into that space. They also convert previously unavailable extended memory above 1024K into EMS memory, which makes it compatible with the EMS 4.0 specification that many large applications require. 386MAX sells for $74.95, 386MAX Professional for $129.95. Reader service no. 24.
Qualitas
7101 Wisconsin Ave., Ste. 1386
Bethesda, MD 20814
301-907-6700

A high-performance graphics card, the ProDesigner II VGA, is due for release by the time you read this. **Orchid** claims that the ProDesigner II is twice or three times as fast as V-RAM VGA cards, and that your rewrite and redraw to the screen will appear "instantaneously." ProDesigner II supports up to 1024 × 768 resolution in 16 colors and has 512K RAM; ProDesigner IIx supports 1024 × 768 resolution in 256 colors and has 1 Mbyte of RAM.

Both support interlaced and non-interlaced monitors, and are downward-compatible with earlier graphics standards. The proprietary Translation ROM allows you to run applications written to earlier standards, and only one card is necessary to run VGA applications. Software drivers are included for AutoCAD, AutoShade, GEM, Lotus 1-2-3, Ventura Publisher, Windows 286 and 386, and WordPerfect. And Orchid claims the card is compatible with nearly all monitors on the market, as well as ISA and EISA machines. Reader service no. 28.
Orchid Technology
45365 Northport Loop West
Fremont, CA 94538
415-683-0300

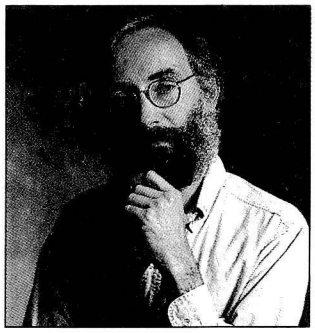
Books of Interest

From **Addison-Wesley** comes the effort of a team of authors that produced *Extending DOS*. Ray Duncan, Charles Petzold, M. Steven Baker, Andrew Schulman, Robert Moote, Ross P. Nelson, and Stephen R. Davis each cover an aspect of DOS programming, from architecture to Windows and DesqView operating environments. It should be available this month, and will retail for $22.95. Reader service no. 30.
Addison-Wesley
Reading, MA 01867
617-944-3700

Prentice Hall has published a comprehensive guide to compiler design and construction in C written by former *DDJ* columnist Allen Holub, who told *DDJ* "The book is oriented towards programming — the programs are the core of the book. It's the only really thorough introduction to compiler design written for programmers by a programmer." *Compiler Design in C* covers such basic concepts as the use of Lex, Occs, Lexical analysis, top-down parsing, code generation, run-time library and back end, and optimization. Four appendices cover miscellaneous support routines, grammars and grammatical transformations, C grammar, and problems with Pascal compilers. Price: $42.20 for 650 pages. Reader service no. 31.
Prentice Hall
Englewood Cliffs, NJ 07632
201-592-2348

DDJ

Three on a Match



As soon as any artificial intelligence technology reaches the point where it actually works, we become reluctant to call it artificial intelligence. That's understandable, but it's also annoying to the people who bring the technology out of the lab and into the market. Their very success makes them look like braggarts if they continue to label their work as they did before it was successful.

Expert systems are a clear cut example of an artificial intelligence technology brought successfully to a number of specialized markets, so successfully that the curse of AI falls heavily on expert systems. It seems pretentious to call them artificial intelligence. So it must be adding insult to injury to suggest that even the term "expert system" is pretentious.

One of the standard "mysteries" of artificial intelligence, posed in most articles and beginning courses on the subject, is the difficulty of simulating a nonexpert. Why, the puzzle goes, is it easier to simulate knowledge and behavior of an expert than to simulate the knowledge and behavior of a child of three? or of a chimpanzee?

The real mystery is why people keep asking the question, "Isn't the answer obvious?" The first synonym for "expert" in Rodale's *The Synonym Finder* is "specialist." Specialization is a narrowing of focus, and there is no mystery at all about why it's easier to simulate behavior in a narrow domain than in a broad one. True, narrowing of focus is not one of the defining features of expert systems, but I'd be willing to bet that it is the one that makes them successful. Perhaps it would be more modest and more meaningful to call expert systems "specialized systems."

Nuke the Innumerate

According to the editors of *CD-ROM End User*, who use the following expression as a running head throughout their useful journal: $\Sigma(Digital\ Data \times Mbytes) * CD\text{-}ROM > E=mc^2$ means, "The sum of megabytes of digital data plus CD-ROM is greater than . . . " (the ellipsis is theirs). On the principle that from a false premise any conclusion can be derived, I suppose, that they can assign any meaning they like to this meaningless sequence of symbols. Perhaps I shouldn't be so hard on the editors; after all, they are only using the expression as techie-looking illustration. But the expression is so egregiously innumerate that it calls into question the technical accuracy of everything in the magazine. To cite one example, should we trust the math of people who don't recognize that they've used different symbols for multiplication in one expression?

My interpretation of the expression is, "When you combine megabytes of digital data on CD-ROM ignorantly, the results are more than likely to blow up in your face."

See Tog Flame

I praised *CD-ROM End User* here recently, and I still think it's an informative publication. I also praised Dave Thompson's *Micro Cornucopia*, which, alas, appears to have published its last issue. Dave, weary of the grind and eager to get on with other things, sold the publication to Miller-Freeman, who probably won't continue *Micro C* as a magazine. Too bad.

Another of my favorite publications, Apple's developer magazine *Apple Direct*, stubbed its toe recently when it announced Apple's new hardware products days ahead of the official announcement date. Another setback for the anti-Perestroika cultural reeducation program at Apple.

What I like most about *Apple Direct*, apart from its timely information on new products, is a user interface design column written by Mr. Tognazzini. (Sometimes he's Bruce, sometimes Bruce "Tog," but usually he's just Tog, which is also his AppleLink address.) Some of Tog's humor may not please either Apple brass or developers: "I have forwarded your link . . . into the very heart of Apple Engineering Land, and, rest assured, we shall eventually do something official." Tog's opinions, of which he has his full share, frequently deviate from the party line, as when he said that he would "rather die than defend this silly business of throwing disks into the trash to eject them."

In that same March issue that preannounced the new hardware, Tog had some sage things to say about user testing on the cheap. Among the radical ideas is that user testing should be done by members of the design team, using no more than three people per design iteration. User testing can show you things you'd never guess about your product and your users. Tog relates an experience in user testing of an in-box tutorial in which one apparently trivial question proved to be the most difficult of all. The "difficult" questions posed no problems for the test subjects, but the "easy" question proved to be the most difficult to phrase unambiguously. He just wanted to find out whether the test subject was using a color or black-and-white monitor. It took six iterations to get the question right.

Michael Swaine

Michael Swaine
editor-at-large

#165 JUNE 1990 $2.95 ($3.95 CANADA)

SPECIAL HYPERTEXT EDITION

Dr. Dobb's JOURNAL

SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

HANDS-ON HYPERTEXT!

Hot Programs For Hot-Links

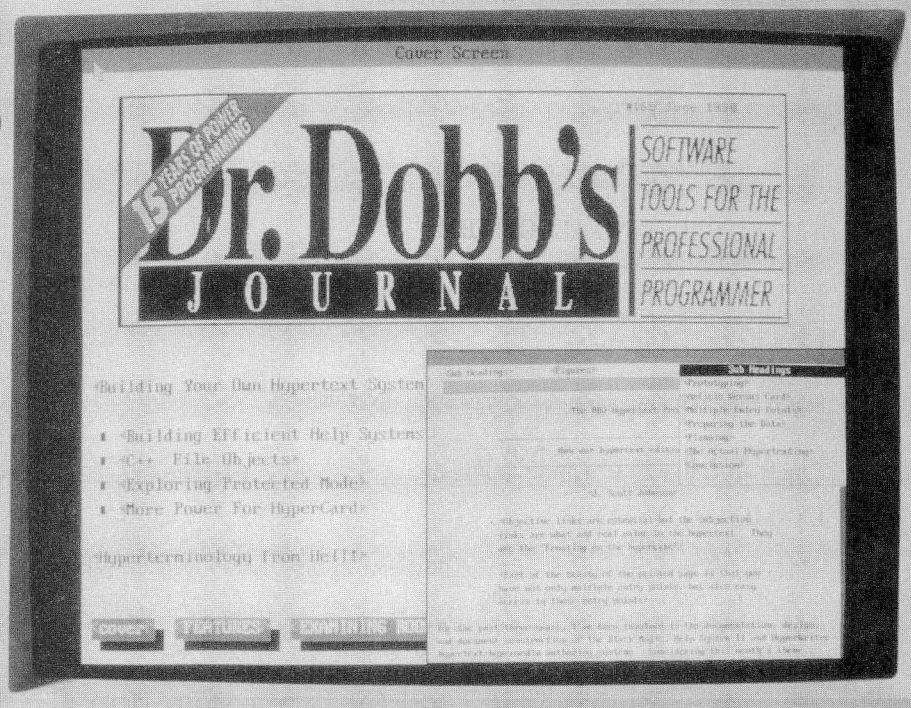
- HYPERTEXT ENGINES
- BUILDING EFFICIENT HELP SYSTEMS
- MORE POWER FOR HYPERCARD

C++ FILE OBJECTS

PUTTING PIXELS IN PLACE

80386 PROTECTED MODE PROGRAMMING

LZW REVISITED



0 38351 16562 8 06

CONTENTS

FEATURES

THE *DDJ* HYPERTEXT PROJECT 16
by J. Scott Johnson
A behind-the-scenes look at the *DDJ* hypertext project by the programmer who put it together.

BUILDING A HYPERTEXT SYSTEM 22
by Rick Gessner
Rick uses Turbo Pascal to build a page-oriented, text-only hypertext system that has embedded "hot-links." His system includes both a screen editor and a hypertext engine.

A SELF-REFERENTIAL HYPERTEXT ENGINE 34
by Todd King
Here's another approach to hypertext systems. This one, written in C, is ideal for context-sensitive help or source-code documentation projects.

BUILDING AN EFFICIENT HELP SYSTEM 40
by Leo Notenboom and Michael Vose
Knowing how help files and a hypertext engine interact is central to effective on-screen documentation.

C++ FILE OBJECTS 50
by Kevin Weeks
The key to efficient programming using object-oriented languages is having a good base class.

A PIXEL ORDERING ALGORITHM 56
by Norton T. Allen
Recognizing the general character of an image early in the display process lets you begin fine-tuning much sooner.

LZW REVISITED 126
by Shawn M. Regan
Shawn enhances this popular data compression algorithm.

EXAMINING ROOM

EXAMINING INSTANT-C 62
by Andrew Schulman
An interactive environment such as Rational Systems' Instant-C takes a lot of the pain out of 80386 protected-mode programming.

PROGRAMMER'S WORKBENCH

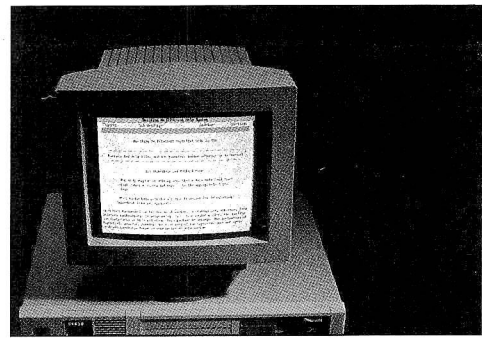
ACCESSING HARDWARE FROM 80386 PROTECTED MODE: PART II 78
by Stephen Fried
Steve argues that the only use of FAR pointers in 80386 code is in operating system kernels. To make his point, he examines ports and interrupts.

COLUMNS

PROGRAMMING PARADIGMS 129
by Michael Swaine
Michael shares different techniques for adding text links to HyperCard.

C PROGRAMMING 135
by Al Stevens
Al develops an indexing technique that is a loose adaptation of the B-tree.

STRUCTURED PROGRAMMING 143
by Jeff Duntemann
Heap fragmentation, and how to manage it in Pascal and Modula-2, are Jeff's topics this month.



DEPARTMENTS

EDITORIAL 6
by Jonathan Erickson

LETTERS 8
by you

SWAINE'S FLAMES 168
by Michael Swaine

PROGRAMMER'S SERVICES

ADVERTISER INDEX 160
where to go for more information on products

OF INTEREST 161
compiled *by Janna Custer*

PROGRAMMER'S MARKETPLACE 164
classified ads

NEXT ISSUE

In July, we'll present our annual Graphics Programming issue, beginning with a supercharged version of Cohen-Sutherland's classic line-clipping algorithm. We'll also look at ways to draw better circles faster and examine Bézier curves.

Putting HyperTheory Into HyperPractice

This issue of *DDJ* showcases the power and promise of hypertext technology, not just in theory but in practice. Thanks to Scott Johnson and the crew at NTERGAID (developers of the Black Magic authoring system, Hyperwriter, and other hypertext development tools), we've taken the exciting (and as far as I know, unprecedented) step of providing you with an electronic "hypertexted" edition of *DDJ*.

What can you do with a hypertext version of *DDJ*? Think about how you usually read *DDJ*: You look at the Table of Contents, spot an article that seems interesting, turn to the appropriate page, and begin reading. When the article makes reference to a figure, you turn more pages. And when you come to a reference to source code listings, you usually flip halfway through the magazine while trying not to lose your place.

Hypertext changes this. As you peruse the Table of Contents on your PC screen, you position the cursor on the title of an article and call up its description, the author's biography, or "turn" to the article itself. When reading the article and encountering a "see Listing One" or similar message, you simply click on the message and the listing appears on the screen.

Getting the hypertext edition of this issue is easy. It comes at no extra charge when you order this month's source code listing disk. Alternatively, you can download the electronic issue from the *DDJ* Forum on CompuServe. (For details, look for the "Availability" notices at the end of just about every article; see page 33, for example.)

Since the hypertext document is built around the Hyperwriter runtime engine, you don't need any special software. You simply "self" extract the file and run it. You will need a PC with 640K of memory, and a mouse makes it a lot easier to use. For more details on the system and an enhanced version that uses bit-mapped graphics, see this month's lead article "The *DDJ* Hypertext Project" where Scott describes how he put the edition together.

Where do we go from here? Consider that if you're like most readers, you keep back issues of *DDJ* as reference material. Wouldn't it be great to have 15 years worth of *DDJ* at your fingertips, cross-referenced and linked on CD-ROM? You could then search for a specific topic or algorithm, bring up all versions of it, examine the code, and then paste it into your application. No, we aren't planning on this right away, but it does give us something to dream about.

All the news about hypertext isn't necessarily fit to print, however. Why? Because the trademark buzzards are at it again, this time circling over the term "hypertext" itself. It seems that a company called Hypersystems S.R.L of Torino, Italy has applied for a U.S. trademark for "HyperText," a stylized version of the term "hypertext." And if the use of ® in their product literature is any indication, the company seems to be claiming a trademark on the stylized term "HyperMedia" too.

Even though Hypersystem's U.S. attorney told me that his client has entered a disclaimer for generic use of the term "hypertext," hypertext and hypermedia developers are nonetheless up in arms, and several, including NTERGAID, Owl International, and Autodesk, have filed oppositions with the U.S. Patents and Trademarks Office.

The rule of thumb is that if a word is either descriptive or generic, the Trademark Office won't issue a trademark. To me, "hypertext" seems descriptive because it simply describes a system of non-linear reading and writing and it is generic because it is commonly used in magazines, books, dictionaries, and other forms of documentation.

If you're as annoyed by this as I am, you should know about a couple of avenues of protest. Software developers who have a hypertext-based product, and therefore a vested interest in seeing the term not enter the Calcutta of trademarks, can file an official "Notice of Opposition," while members of the general public can send a "Letter of Protest."

If you believe that a trademark on the term "hypertext" would damage you or your product in some way, you should file a Notice of Opposition with the *Trademark Trial and Appeal Board.* You'll have to pay a fee and you must "set forth a short and plain statement" stating how you would be damaged by the trademark.

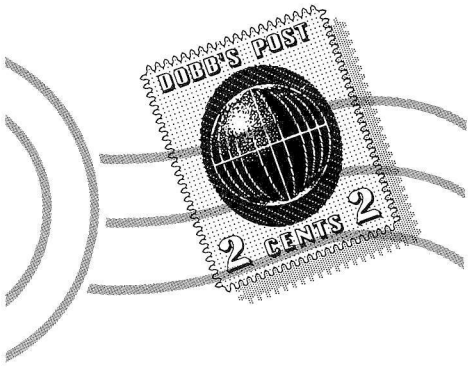
Letters of Protest should be sent to the *Office of the Director of the Trademark Examining Operation.* The purpose of your letter should be "to bring to the Office facts that could affect or prevent the registration of the mark . . . . The letter must contain proof and support of the information . . . " before your protest will be considered.

For more information about either the Opposition or Protest, contact the U.S. Department of Commerce, Patent and Trademark Office, Washington, DC 20231.

In closing, let's not forget that Ted Nelson, now at Autodesk, coined "hypertext" in 1963 while Doug Englebart, now at Stanford University, defined the technology even before that — and neither found it necessary to trademark the word.



Jonathan Erickson
editor-in-chief



Hidden Secrets

Dear *DDJ*,

The S-CODER algorithm Robert Stout presented in the January 1990 issue of *DDJ* seems reasonable, but the programs using it (Listings One and Four) don't give adequate protection against Kerckhoff's superposition. This 19th-century technique is mentioned in the cryptographic hobby literature (perhaps in *Secret and Urgent* by Fletcher Pratt or *The Codebreakers* by David Kahn). The method is not very computationally intensive, as one might expect given its age.

Consider Listing One. Suppose NF files are encrypted with the same key. Since cryptext[] does not depend on the plaintext, all the files are encrypted by XORing with the same byte stream X[1], X[2], . . . If P[i,j] is the *i*th byte in the *j*th plaintext file and C[i,j] is the *i*th byte in the *j*th ciphertext file we have

C[i,j] = P[i,j] XOR X[i]

If we form the set of bytes C[i,j], j = 1, . . . NF for some fixed *i*, all these bytes are encrypted by XORing with the same byte. We can readily find the value of X[i] that decrypts this set of bytes most resembling plaintext. For example, if we assume ASCII plaintext, the space character will be overwhelmingly the most common (ordinarily) and X[i] = (most common byte in the set C[i,j], j = 1, . . . , NF) XOR <space> will be right most of the time. We can do this for each value of *i* separately without worrying about the recursion relating the X[i]s.

For Listing Four the first thing to observe is that the transposition is keyed with a sequence of random numbers whose seed is in the cryptanalyst's possession. He can thus remove the transposition at will. The other problem is that *crypt_ptr* is initialized to (file length) mod (key length). The file length is not concealed from the analyst, however, and the key length might well be less than 20. With several hundred files encrypted with the same key in his possession, the analyst can assume a key length and then group together sets of files whose lengths do not differ modulo the key length. He can then still use the superposition method on these sets of files. If he has assumed the wrong key length he will quickly notice that the sets of bytes C[i,j], j = 1, . . . , NF will have frequency distributions that are too smooth.

I suggest two changes. First, the random number seed should depend on cryptext[] so the analyst will need to deal with a nontrivial transposition. Second, the initial values of the bytes in cryptext[] should be modified using the sum of many or all of the bytes in the plaintext so that the chances of many files having identical key streams X[i] will be negligible. This sum of bytes can be put openly in the file header along with the length, or some more devious method chosen.

Stewart Strait
San Diego, Calif.

Bob responds: First of all, thank you for your comments. As suggested in the article, S-CODER isn't necessarily an end in itself, but an embeddable encryption engine. The final listing in the article begins to suggest ways to create more secure applications, although the example I used is subject to fairly straightforward cryptanalysis. This, incidentally, was deliberate, leaving open the option of embedding S-CODER in a more secure wrapper as a challenge to critics.

Aside from the principle that S-CODER should be considered more of a building block than anything else, the two primary points I'd like to stress are that:

1. The unembellished S-CODER engine was not suggested as being particularly secure when faced with known plaintext analysis, but is rather more suitable for short-term security or unknown plaintext applications. In this sort of application, the only requirement would be that the key is longer than the longest item of plaintext, which might be known by implication (for example, words like "the," a company name, etc.);

2. A truism of data security is that a theoretically secure method known to everyone (e.g., DES) is often less desirable than a theoretically less secure method (e.g., an enhanced algorithm using an embedded S-CODER engine) which is not known. S-CODER buried within an unknown block transposition cipher as suggested in the article, or something equally (or even more) obscure presents the cryptanalyst with the immediate problem of classifying the algorithm before being able to break it. Knowing only that S-CODER is buried in there somewhere is of little help since S-CODER is a stream cipher and any enhanced application will randomize the serial dependencies of the S-CODER encryption.

Addressing your specific objections, cryptext[] is only independent of the plaintext during the first pass through it. The ability (suggestion, actually) to set crypt_ptr *at some arbitrarily random starting point within cryptext makes the kind of analysis you suggest much more difficult. If we assume that the initial value of* crypt_ptr *is determined by some unknown feature of the plaintext, the relationship between* i *and* j *in your example will be randomized. In your second point, you're absolutely correct that the implementation in Listing Four is flawed in that the information used to provide the additional security is openly available to the cryptanalyst. Any conceivable method used to conceal the file length would remove this objection. Once this vital piece of information is concealed, the next step is to modify the actual function within which S-CODER is embedded. Ideally it should be fast, as is the block transposition cipher shown in the listing, but the exact method should be known only to the implementor. This is what I meant above when I said that an unknown method may offer more security than a better, yet known, method. The real value of Listing Four is in showing that such methods needn't sacrifice any significant performance over the raw S-CODER algorithm to offer significant levels of security. One of the purposes of the article was to encourage people to think of how to write practical software rather than worry about theory so much. I welcome your contribution and will consider your suggestions in future applications.*

Fighting Software Patents

Dear *DDJ*,

Recently your magazine lightly touched on the subject of software patents. In the April 1990 issue of *Technology Review* (Building W59, MIT, Cambridge, MA 02139), Brian Kahin presents a frightening view on the subject. Even if part of what he envisions comes to pass, software as an entrepreneurial business could be destroyed. I recommend this article to all who work in the art and science of software.

(continued on page 12)

(continued from page 8)

I do not understand why we have put up with this kind of silliness. An aroused software community (users and authors) using the networks (USENET, BITNET, CompuServe), bulletin boards (both local and national, like yours), and their personal word processors can make a Very Loud Noise. These communications resources can be used for information exchange and coordination. If everyone then processes one letter a week to some politician somewhere, the very idea of software patents can be quickly forced out.

DDJ is a technical magazine. The whole idea of software patents seems to have arisen from technical naivete in legal and political circles. We need to have solid technical, intellectual, and philosophical arguments against software patents circulated amongst ourselves and in any circles that are affected. I urge *DDJ* readers to carry out this idea and start a word processing campaign.

Jim Iwerks
Rio Rancho, New Mexico

Pick-A-Fight Interfaces

Dear *DDJ*,

I am surprised that someone with as much computer experience as Bob Canup would write such a one-sided and narrow minded article as "Pick-A-Number Interfaces," which appeared in the February 1990 issue of *DDJ*. Mr. Canup's entire argument seems to be based upon a poorly written mailing list program that he had had experience with. A properly written application level interface should be intuitive enough for any new user to feel comfortable with and should also assist the experienced user into getting the job done as quickly as possible. The ultimate goal in assisting a new user is to make the application as "real world" as possible. An example of this would be to use the letter "B" as the backup option. The relationship between the letter "B" and the word "backup" is simple and intuitive and lends itself to concepts the user already understands. Bob proposes forcing the user to learn a new and unfounded relationship between the number six and "backup."

Webster defines the word "friend" as "A person who one knows well and is fond of," this is exactly the type of response that a programmer wants to achieve with a user interface. Modern UI techniques help users feel like they are working with objects they know and understand. A piece of paper can be represented by a window; just as someone may lay another form down on their desk, they can lay down a new window on the screen. A windowed interface can also shield the user from absorbing information they do not presently need and at the same time graphically represent their "position" within the application. Although pointing devices such as a mouse can be a hindrance to the experienced user, they are often helpful to the new user by allowing them to operate an application without having to know a predefined set of relationships between the application and keyboard. A mouse user can simply click their pointer on the word "backup." It may take slightly more time to execute the command, however, the knowledge required to select an option is reduced to understanding the operation of the pointing device. As the user becomes more proficient at their own pace they may begin to use the keyboard correlation commands. On the other hand, the keyboard user must not only know how to operate the keyboard but also must learn relationships between keys and options in the application before they can operate it at all. Modern interface techniques provide a much more supportive interface for the user. Mr. Canup makes the assumption that the use of these tools is the problem, when really it's their misuse. I may be able to saw a piece of wood in half using a power drill; although there are much more effective means, does this mean that the power drill is a useless frill? Obviously not! If used properly, a power drill can save a great deal of time in a project. I am certainly glad that Mr. Canup is not the guiding light behind innovative and forward thinking user interface design.

David Bennett
CompuServe: 74635,1671

Standards Revisited

Dear *DDJ*,

Since you gave Tex Ritter, P.E., a full page, I hope you will print a few words of mine. Tex Ritter, P.E., takes a couple of thousand words to tirade about the standards process ("Letters" *DDJ*, February 1990), but reveals that he has forgotten those portions of the scientific method he learned.

He didn't bother to learn that the "draft standard" is not an early, highly changeable document but the next-to-last step in acceptance and is nearly frozen in place with people using it as a basis for software.

He spent many hours responding to everyone's comments that he received in June '89 during the 15-day response time and considered the response in September "appallingly condescending." Again he didn't do his research. At the stage he did all that work, he only had the right to respond to his own previous comments. Other people merely complain that it takes too long to search for the changes that caused their own previous comments.

His greatest complaint seems to be that he wanted a lot of features that violate the Pascal philosophy of rigidity and easy teaching/grading (which is why I don't like Pascal) and he wanted strong engineering math types. In the latter case he should be using Fortran, except he likes the lazy ease-of-use of never-standard Turbo Pascal.

After stating that voting is not the way to select a standard, he decries the "back room maneuvering" that put together the standard. He is ignoring the hundreds of hours put in to create a consensus document that could be voted on. It isn't democracy, it is consensus building among people who are scattered across 3.6 million square miles. Of course, Ritter would probably object if the meetings were held anywhere but at his Austin home.

All I know about the standards process is the descriptive articles that have appeared in *Dr. Dobb's*, *Unix Review*, and *C User's Journal*. Obviously, I know more about the process than Mr. Ritter, who tried to participate but effectively entered the pool with a belly flop. I hope that he will research his next project (and his P.E. work) in a more professional way, so he does not waste his time.

Mike Firth
Dallas, Texas

Great Minds . . .

Dear *DDJ*,

It is most interesting that Michael Swaine has created an article on the War on Bugs as a pun on the national War on Drugs ("Swaine's Flames," *DDJ* February 1990). I enjoyed the sarcastic humor and appreciate the work that went into making that article fly.

About a year ago, I too noticed that bugs and drugs had many similar qualities (for example, their tenacity and their ability to break down morale), and decided to include the following in the origin line on my FidoNet BBS:

*Origin: Programmer's Oasis /
919 – 226 – 6984 — Say NO to Bugs!
(1:151/402.0)

Strangely enough, that is the only pun that was not utilized in the article! Oh well. Thanks for an entertaining column in a highly enjoyable magazine.

Chris Laforet
Graham, North Carolina

(continued on page 14)

(continued from page 12)

Ritter Redux

Dear *DDJ*,

After reading Tom Turba's response (April 1990 "Letters") to my published letter (February 1990), I have the feeling that he missed his true calling: He has a great future in politics. Although the individual facts he cites may well be true, the impression he leaves is deceptive. Essentially he says: "We have rules which protect your interests; if you did not take advantage of them, it is your own fault." But it is one thing to leave the doors open to participation, and quite another to announce where the doors are, and what is on the agenda. A casual half-hour with my old programming magazines turned up no less than eleven articles over the past two years on or about ANSI C standardization issues; in contrast, I found no articles on ANSI Extended Pascal. It appears that Pascal users have had no notification whatsoever of the Extended Pascal effort.

Tom says that he invites our "participation," but there is participation — where one may have the chance to speak, only to be ignored — and PARTICIPATION — where one's concerns are satisfactorily addressed. I obviously did participate, and certainly did not approve of the Extended Pascal specification, and yet Tom tells us that consensus and even unanimity was reached. Such participation reminds me of the communist party — all comrades are equal, but some are on the central committee. I guess if you don't physically walk through the doors you must not be participating, and maybe if you do, you get to be a "visitor" or "observer." Clearly, this scheme was not designed to increase participation.

Apparently Tom has now gotten a clue: A possible future project (object-oriented extensions) may be announced to the user community. Great, Tom, but the problem is the current specification, not extensions to it. As I see it, Extended Pascal was developed without significant involvement from the user community, so the process, as arduous as it may have been, was fatally flawed. The resulting "standard" is invalid. It is time to throw this one away, and start on one we can all accept.

I understand that those may be chilling words to some people. To the individuals who have participated in the many long debates which may have taken place, this might seem like the destruction of all their work. Yet if persuasive arguments exist for the various decisions, they should again prevail; all that would be lost are those political victories which were not technically warranted. To software companies who see standardization as a marketing tool, building consensus within the user community must seem like a costly and unnecessary delay, but what good is a standard if most users reject it? As important as these views are, they pale beside their effects. For example, high school students are being subjected to "Standard Pascal," because they will eventually take placement exams which use this dialect. Without getting into whether a likely pencil-and-paper exam can have any bearing on the interactive use of a language within a fast development environment, it is clear that even a marketplace-rejected standard can make its obnoxious presence known.

Tom simply failed to address the biggest problem with the Extended Pascal specification: It is not really a specification at all. Look at it this way: Suppose you are a programmer in a small company hoping to write portable programs for several different platforms. After buying or licensing a "standard" compiler for each, your "portable" programs do not compile on one or the other. Now, given the Extended Pascal specification, will you be able to decide which compiler is at fault?

Pascal is a STRUCTURED programming language. In contrast, the Extended Pascal specification is "spaghetti code" throughout. In the version I bought, terms were poorly defined, making the specification inherently ambiguous. The spec did not stand alone, but appeared to rest on previous work, perhaps upon unnamed compilers on unnamed machines. The shear complexity of it led me to question whether The Committee had ever dealt with an actual complex Pascal program, for if they had, they would have been used to partitioning complexity into understandable chunks. The specification is a turkey, not just because it does not have the right things in it, but because it is virtually impossible to deeply understand what is specified, and therefore impossible to understand the ramifications of the various features in a final product.

But you don't have to take MY word for it: Tom invites your participation, which is great! Take him up on it! If you think my comments are mostly sour grapes, well, just get Tom to send you copies of all the comments and responses for the past two rounds, and make your own decision! If you think that the Extended Pascal specification cannot possibly be as bad as I say, then Hey! Get Tom to send you a copy and read it! If the spec looks fine to you, tell me I'm all wet. If you resent the idea that major standardization decisions gave gone on without widely available printed discussions on the issues, let Tom know, let ANSI and ISO know, and also let your compiler vendor know.

Terry Ritter, P.E.
Austin, Texas

Rhealstone Suggestions

Dear *DDJ*,

I have just finished reading through the April 1990 issue of *Dr. Dobb's* and was very pleased to see that you had an article with an actual implementation of the Rhealstone benchmark. May I suggest a couple of additions which I would like to see added to the Rhealstone benchmark:

1. Intertask message latency between tasks that are on different nodes of a network. This metric is important to developers who work on networked real-time control system applications.
2. Message/datagram throughput between tasks as a function of message size. This should be measured between two tasks that are on the same node and also between tasks that are on different nodes.

Nick Busigin
Stratford, Ontario
Canada

Animation Algorithm Update

Dear *DDJ*,

After reading the letter sent by Peder Jungck ("Letters," April 1990), I felt that some clarification was necessary. In his letter, Mr. Jungck stated that a faster algorithm would be to separate the pixel data into logical planes before the data is moved to the EGA's physical display planes. Mr. Jungck was apparently assuming that my sample program translated the initial one byte/pixel format. This is to maintain a rough compatibility to some future VGA sprite driver. This first part is only run once. The second part uses the format that Peder suggested. It is the structure that is used every screen refresh. Although I'm sure Mr. Jungck's program is a peach, it does not use an algorithm different from what I provided.

Rahner James
Sacramento, Calif.

DDJ

We welcome your comments (and suggestions). Mail your letters (include disk if your letter is lengthy or contains code) to DDJ, *501 Galveston Dr., Redwood City, CA 94063, or send them electronically to CompuServe 76704,50 or via MCI Mail, c/o* DDJ. *Please include your name, city, and state. We reserve the right to edit letters.*

The *DDJ* Hypertext Project

How our hypertext edition happened

J. Scott Johnson

For the past three years, I've been involved in the documentation, design, and document construction of the Black Magic, Help System II, and HyperWriter hypertext/hypermedia authoring systems. Considering this month's theme, preparing a hypertext version of *Dr. Dobb's Journal* using HyperWriter seemed an ideal, yet challenging, project. This article describes the steps I took and the decisions I made. While some of these steps were unique to the task at hand, overall this is typical of most hypertext projects.

Prototyping

My first task was to prototype the "look and feel" of the hypertext version. I took the September 1989 issue of *DDJ* and constructed two prototypes of the document — one article-based, the other card-based. The major distinction between the two can be summed up in one word: scrolling. Article documents "scroll" up and down the screen while card-based documents "flip" from one screen to another. An article-based document can have unlimited length in any of its nodes, while a card-based document is a collection of screen-sized nodes linked together.

Initially, I didn't know which approach was more appropriate. Though I like card-based hypertext documents and understand that users often find them easier to navigate through, I first opted for the article metaphor. For one thing, card-type documents are nearly impossible to use without a mouse. Mice, though common, are not everywhere; with an article-metaphor system, keyboard navigation can be efficiently employed. In addition, article-metaphor hypertext tends to be significantly easier to create than card-based hypertext, which requires you to "chunk" the material across multiple cards. After evaluating the two metaphors and how readers might interact with them on the screen, I ended up using elements from both, as described in the next section.

Another decision I had to make during the prototyping stage involved the size of the document. The goal was to fit everything — document, runtime, and source code — on one or, at the most, two floppy disks. This conflicted, however, with my desire to create the best possible document, a full issue of *DDJ* on disk. I compromised by creating two versions. The version shipped with the listings disk is a bare-bones hypertext document containing articles and source code. To save disk space, the graphics for this version were created with the extended character set. The other version, however, is a complete issue of *DDJ* — articles, graphics, letters, and more. Directions on how to obtain both documents are at the end of this article.

Preparing the Data

One of the first steps in creating this hypertext was data preparation. I was supplied with ASCII text files extracted from *DDJ*'s typesetting system, which could be readily imported into HyperWriter. This made the text-end of things relatively easy. Graphics, however, were a different story.

Most of the graphics were supplied in a Macintosh format. This meant converting the graphics from MacDraw to a MacPaint bitmap and bringing that bitmap to the PC with a Copy II PC option board. Next, I converted the MacPaint bitmap to a .PCX bitmap format, correcting the aspect ratio of the Macintosh bitmap to appear correctly on the PC's screen. Finally I converted the corrected .PCX bitmap into HyperWriter's compressed bitmap format to minimize disk space. Although this sounds like a lot of work, it is mostly just tedious. In addition to converting the graphics images, I had to construct versions of each graphics file with the extended ASCII character set. Once these tasks were done, the real work of building the document could begin.

Planning

When all the data was in the computer and in machine-readable formats, I was ready to start building the document. Right? Not quite. I've found that before creating any large hypertext document, a small amount of planning makes

Scott is the president of NTERGAID, developers of a variety of hypertext and hypermedia tools. He can be reached at 2490 Black Rock Tpke., Suite 337, Fairfield, CT 06430, 203-368-0632, or on BIX as s.johnson, and on CompuServe at 75160,3357.

worlds of difference. An early constraint was to optimize the size of the hypertext document to accommodate HyperWriter (the toolkit I used), which currently works only with hypertext documents that can be contained within memory.

To provide fast access to the contents of documents that are actually spread across several files, I duplicated the Table of Contents cards in each hypertext document. This eliminated disk delays caused by simply accessing the Table of Contents. Now, disk delays occur only if the article being accessed is actually stored in another hypertext document. To help me build these documents that all share a common Table of Contents, I first created a "kernel" document that contained only the Table of Contents cards. I then used the kernel document as a base from which to create the rest of the hypertext documents.

The Actual "Hypertexting"

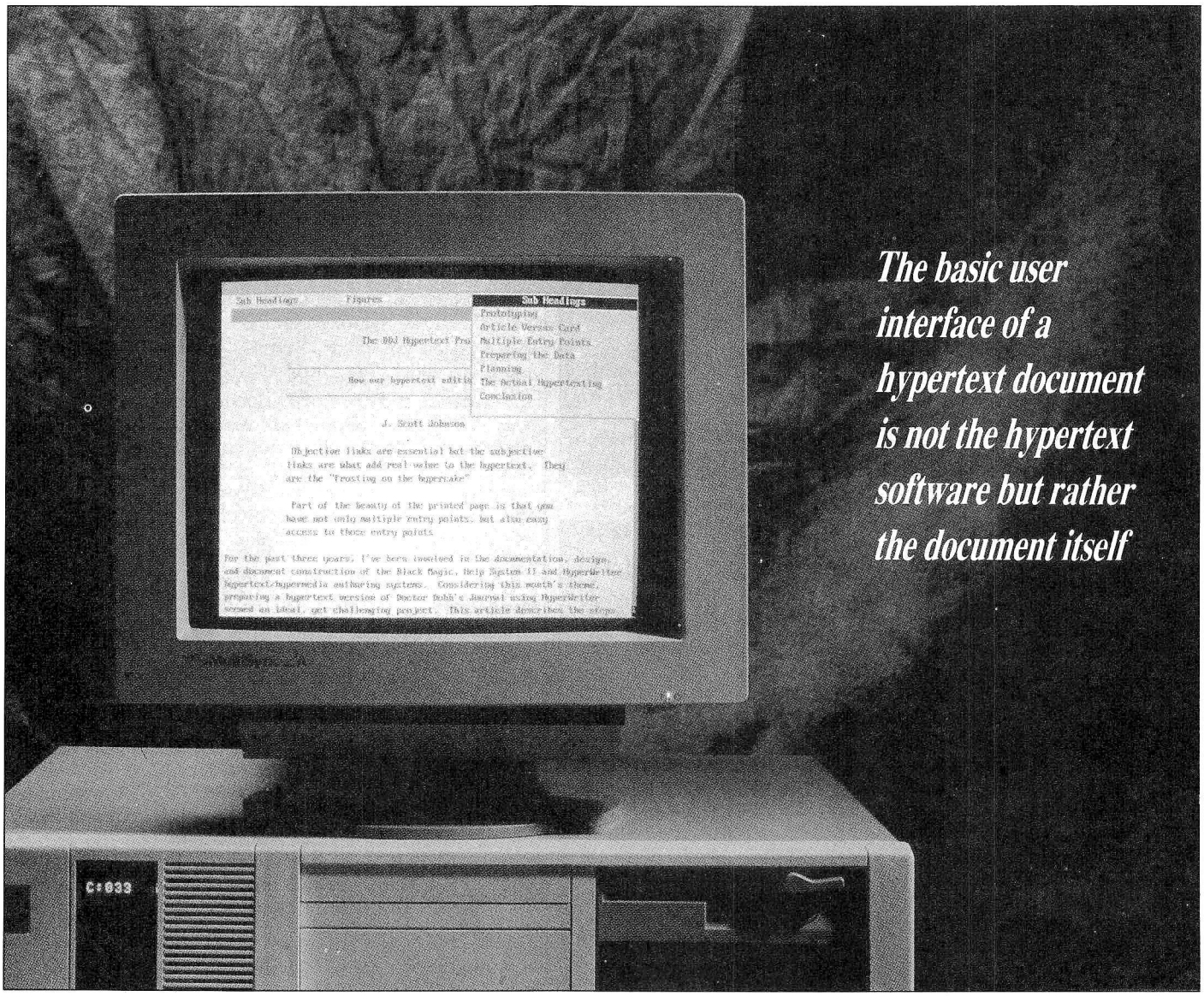
Once I finished building the framework of the hypertext documents and formatting each article, the real work of "hypertexting," or creating the cross-referential links, could begin. In the hypertext field, there is a growing distinction between what are called "objective" and "subjective" links. An objective link is a link from the table of contents to a particular article. A subjective link, on the other hand, is an editorial link that cross-references relevant material or makes a comment on existing material. What we've found so far is that while the objective links are essential, the subjective links are what add real value to the hypertext. They are the "frosting on the hypercake" (so to speak). The major example of subjective links in the *DDJ* hypertext are the links to the source code files. These links allow readers to dynamically view both the articles and the source code. Building the subjective links, beyond the source code links, required analyzing the material and then creating links where appropriate.

Using the *DDJ* Hypertext Document

When you first open the *DDJ* document, you'll see a screen like that in Figure 1, which represents a *DDJ* cover. This card-based format offers access to the different sections of the Table of Contents. Select any of the links along the bottom of the screen to access a section of the Table of Contents. Selecting one of these links yields a Table of Contents card like that in Figure 2.

You can select any of the links from a Table of Contents card and access an article. Articles are presented in article-metaphor format. A sample article from the issue is shown in Figure 3.

I made an interesting observation on the nature of hy-



The basic user interface of a hypertext document is not the hypertext software but rather the document itself

pertext during the design of the article-metaphor parts of the document. The basic user interface of a hypertext document is not the hypertext software but rather the document itself. The arrangement of a hypertext document — that is, its node and link structure — forms the bulk of the user interface that the reader encounters. The only time a reader steps outside the "document as interface" is when he or she elects to use the features of the hypertext runtime itself. To further enhance the document as user interface, and to lessen the learning curve for existing readers of *DDJ*, I adopted as many of *DDJ*'s standard formatting conventions as possible. See Figure 3, which shows the beginning of one of *DDJ*'s feature articles. The one major difference in formatting conventions is the substitution of different colors in the hypertext version for different fonts in the magazine.

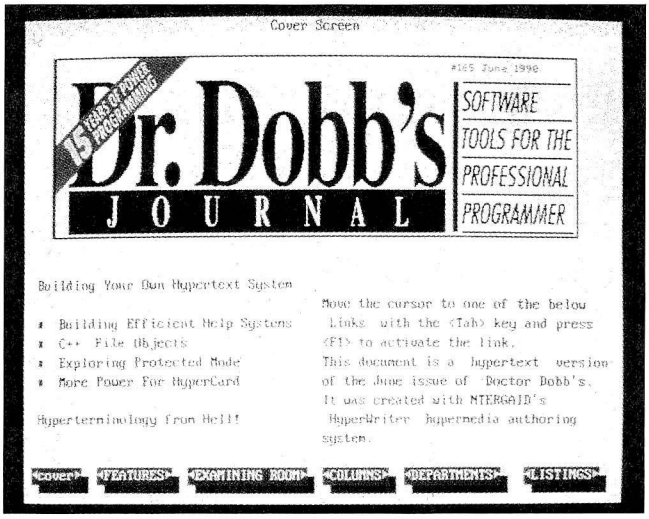


Figure 1: *The* DDJ *document hypertext cover*

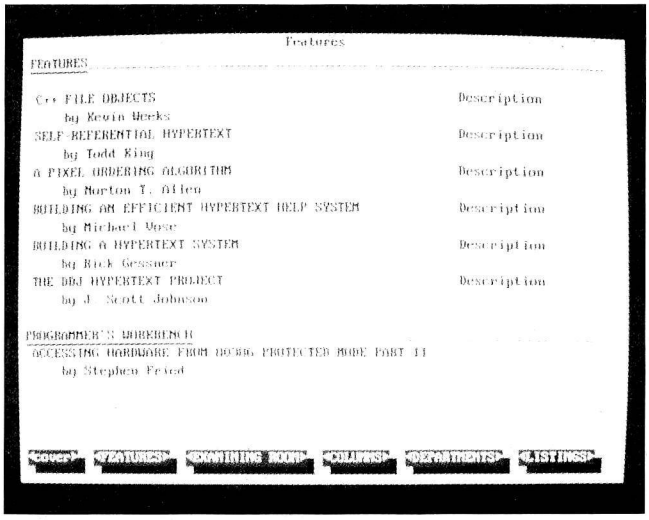


Figure 2: *The Table of Contents card*

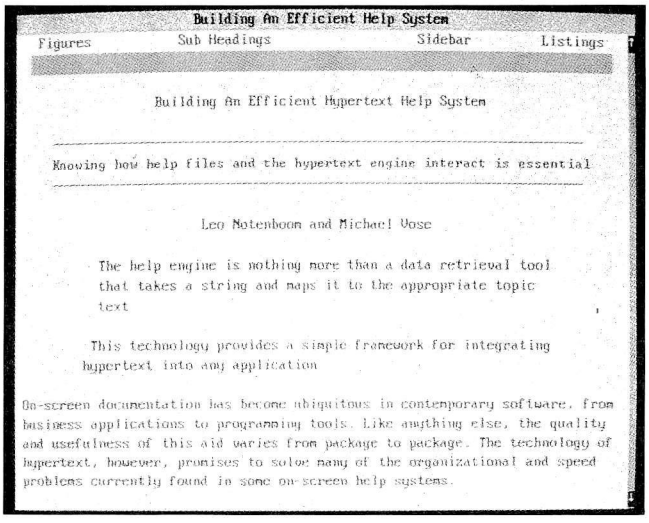


Figure 3: *Sample article in the* DDJ *hypertext edition*

Objective links are essential, but the subjective links are what add real value to the hypertext. They are the "frosting on the hypercake"

Multiple Entry Points

Entry points provide a place to start reading or browsing. Part of the beauty of the printed page is that you have not only multiple entry points, but also *easy* access to those entry points. Some examples of commonly used entry points include subheadings, figures, and sidebars. To access any of these entry points on paper, you flip through the pages of a magazine article or book.

Although hypertext allows the same sort of entry points as paper does, those entry points are often inaccessible because they are off-screen. To address this problem, each article in the *DDJ* document has a top line similar to that shown in Figure 3, allowing easy access to all of the figures, subheadings, and tables (the different entry points into the article).

Conclusion

From the above description, you may get the impression that creating the *Dr. Dobb's* document was a lot of work. It was. We initially allotted six weeks for the job. The final version was produced, working part-time, in just under three weeks. Overall, much of that time was devoted to conceptual activities such as design and planning. Building the actual document was relatively easy with the exception of creating the subjective links — that took real analytical effort.

Availability

The smaller version of this hypertext document is available with the listings disk. The enhanced version can be downloaded from the *DDJ* Forum on CompuServe or you can get it directly from me for a small handling fee ($15.00), along with an expanded version of this article.

The bare-bones version of the document only requires a monochrome display card and 640K of RAM. The enhanced version requires a CGA, EGA, HGA, or VGA system, 640K of memory, and a hard disk. A mouse is recommended. You do not need any special software to use the hypertext system since it is built around the HyperWriter runtime engine. You simply extract the self-extracting file and type *DDJ*. Specific instructions are provided on disk.

DDJ

Building A *Hypertext System*

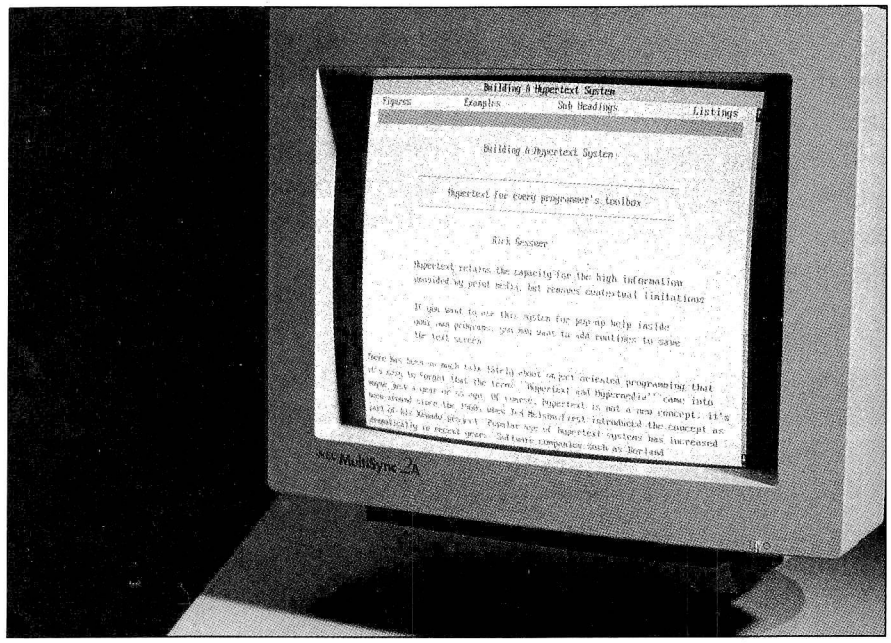
Hypertext for every programmer's toolbox

Rick Gessner

There has been so much talk lately about object-oriented programming that it's easy to forget that the terms "Hypertext and Hypermedia" came into vogue just a year or so ago. Of course, hypertext is not a new concept; it's been around since the 1960s when Ted Nelson first introduced the concept as part of his Xanadu project. Popular use of hypertext systems has increased dramatically in recent years. Software companies such as Borland International and Microsoft, for instance, have designed interactive help components for their products using the hypertext model.

The advantage of hypertext becomes evident when you consider the inherent disadvantages of the traditional information medium — the printed page. The presentation of information in written form is explicitly linear. Structurally, details embodied by the text are intended to move from the general to the specific, while still remaining focused on a given subject. As a result, the medium is confined and contextually inflexible.

Hypertext retains the capacity for the high information content provided by print media, but removes contextual limitations. In contrast to the objectively linear nature of the printed page, a hypertext system grants the user "subjective linearity." This means that rather than having to follow the explicit course of a document, the user can choose to explore an exponential array of contextual alternatives. In other words, the user is free to explore any information path in the document. From the programmer's perspective, this approach removes the burden of deciding the order in which large quantities of information (such as those found in help systems) will be presented. As long as the data is related contextually within the hypertext system, the user can access the information in the manner that he or she sees fit. (One less headache for us!)



The purpose of this article is to demystify hypertext systems, and to show you how to take advantage of the hypertext paradigm in your own programs. Along the way, we'll also touch on a couple of general-purpose programming tricks.

And Away We Go...

A wise old programmer once taught me that good programming is as much about the process of building system tools (providing tools for ourselves) as it is about the process of building applications (providing tools for others). The hypertext system presented here was designed to provide both kinds of tools: A system tool for creating and editing "hyperdata files" (the data files used in a hypertext system), and an application tool that allows users to browse through those files.

The hypertext system presented in this article is a page-oriented, text-only

Rick is developing artificial life software at Anthrobotics in Tempe, Arizona. He can be reached at 1311 W. Baseline #2145, Tempe, AZ 85283, or on CompuServe at 71521,1226.

(continued from page 22)
system with embedded "hot-links" to related information. (The terms "page" and "screen" are synonymous.) By activating any of these hot-links, the user can conveniently move from one item to another, and from one help page to another. Consider the screen in Figure 1. A hypertext page can be comprised of any text that you want in the page. The embedded, highlighted regions (shown in outline) represent the hot-links, which are available to the user as a method of navigation. By using the arrow keys to move the selection bar (shown in inverse video) between hot-links and then pressing the Return key, the user can activate any page.

You might have noticed that the word "Directory" appears twice on the page shown in Figure 1. The first time this word appears, the context of the word represents a noun that means "The logical grouping of files within a subdirectory." The second time the word "Directory" appears, it is used as a verb that means "Display the contents of the current directory." The same word appears twice, in a different context each time. From the user's perspective, the context is everything — it is the basis for the direction that the user takes while using the system. From a systems perspective, the contextual meaning of hyperdata files is nonexistent; it's all just arbitrary data. Herein lies the beauty of hypertext. A user's perception of context can be exploited by using only elementary computing techniques.

Given an Infinite Time, and an Infinite Number of Monkeys . . .

There are probably as many different ways to write a hypertext system as there are ways to pronounce "Bjarne Stroustrup." Obviously, you need a data structure to store the textual data that will eventually be displayed, as well as a way to store and manipulate hot-links. Upon first consideration, a programmer might build data structures that look like the structure shown in Example 1.

Of course, there is nothing manifestly wrong with this approach. (At least that's what I told myself when I did it.) Well, almost nothing wrong. A key disadvantage to this approach is that you must predefine how many hot-links a page can have, and then either preallocate that number of links per page or create a linked list to store the links. The use of an array results in the inefficient use of memory. If you allocate too few elements, you limit the number of hot-links per page; the allocation of too many elements wastes RAM. The use of link lists is not much better, because these lists require the use of additional routines to manage both the linked lists themselves, and disk I/O to and from the linked lists. What's more, if a doubly linked list is utilized, then each node requires 8 bytes just for list pointers — and 8 bytes is more memory than each hot-link record requires. Either way, you have unnecessary overhead.

Another disadvantage with the approach just described is that you have to create two separate editors if you want the system to be fully interactive. The first editor allows the user to edit the text that appears on each page, the second would be used to allow the user to manipulate the hot-link data structures. Moreover, the additional complications of the process of managing the hot-link data itself arise. For example, should the hot-link keywords that appear on each page be unique? If so, then the editor must be able to prevent the user from entering duplicate hot-link keywords. Eventually, it becomes apparent that there must be a simpler way to implement the system.

The answer? What if there were a way to embed the hot-links into the text itself? If this were possible, then the only factor limiting the number of links per page would be the size of the page itself. Another advantage of the technique of embedding hot-links into the text is that you would only need one editor. The only requirement of this editor would be that it must be able to distinguish hot-links from normal text. Other than that, a simple line editor would suffice. (Just wait, you'll see.)

The inner working of this type of system is so simple and elegant that it's almost laughable. The entire program — support routines and all — is just over 500 lines of code. It just goes to show you that the best solution is not always the most difficult to produce.

The system is divisible into two components, each of which is incorporated into a unit called "HyprText.Pas" (Listing One, page 86). The first component handles all of the user interaction required to set up the hyperdata files; this component is basically a screen editor. The second component is called a "hypertext engine," this component combines the information from the hyperdata files together with user input in order to navigate the pages of hypertext.

Already mentioned, the system works by combining the textual data that ap-

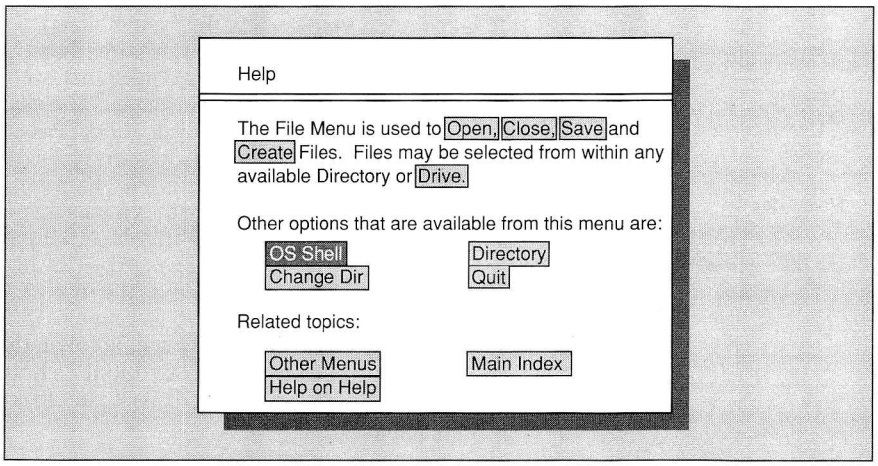


Figure 1: *Sample hypertext dialogue*

```
Const   ScreenWidth = 60;
    LinesPerScreen = 15;
    MaxHotLinks = 25;  {Or any other number you want}

Type       {heres a place to store the screen text}
    ScreenTextBuffer = Array[  1..ScreenWidth,
                    1..LinesPerScreen] of Char;

    HotLinkRecord = Record
        Startpos,        {col pos where link occurs}
        LineNum  : Integer; {row/line number where link occurs}
        LinkPage : Integer;  {page number to activate for link}
    end;

         {now put it all together}
    OnePage = Record
        TheText  : ScreenTextBuffer;
        TheLinks : Array[1..MaxHotLinks] of HotLinkRecord;
    end;
```

Example 1: *A typical data structure*

(continued from page 24)
pears on a given hypertext page with the links that interconnect one page to another. An example of how it's done will make the trick obvious.

Let's assume that the user has typed the lines shown in Figure 2 for a given hypertext page, using our built-in hypertext editor. The characters typed by the user are stored in the *Help_Record* data structure, which is defined as listed in Example 2.

As you can see, the *Help_Record* data type is very sparse; in fact, it's an array of strings. The reason for using a record structure is to facilitate disk I/O, and nothing more.

Here comes the linking trick. Let's presume that a hypothetical user wants to create a hot-link for the term "OS-Shell" (Figure 2) to page 5 in our hyperdata file. (Setting and removing a link is accomplished by pressing the F2 key.) The pseudocode description of the algorithm for placing a link is shown in Figure 3. The process begins by determining if the cursor is currently positioned on a valid (non-space) character. If so, then the next step is to find the first and last characters in the current phrase. The high bit of each character in the phrase is then set, by adding *$80* to the value of each character. The last step is to add a "hot-link token" to the end of the phrase. A hot-link token is comprised of a null byte followed by the hot-link page number.

Returning to our example, the user has decided to create a link on the phrase "OS Shell" to page 5 in the hyperdata file. Before the link, the characters on line 8 look like those shown in Example 3. After the link is performed, the high bits are set, and the link information is appended to the end of the phrase, line 8 looks like the code in Example 4.

Notice that the ordinal value of each of the characters in the phase has been incremented by *$80*. Also, notice that a hot-link token has been added to the end of the phrase (*$00,$05*). A null (*$00*) value is used as a delimiter, and it is immediately followed by a byte that contains the link page number. The use of a byte to store the page numbers limits the system to a maximum of 255 pages of help, which is equal to the greatest value that can be stored in a single byte. It is possible to eliminate this limitation by promoting the byte to a word or a *longint*, as long as you watch out for nulls. (Nulls are used as the unique hot-link delimiter.)

You may be wondering why the system both adds hot-link tokens and sets the high bit of linked terms. There are two reasons. First, setting the high bits simplifies the task of detecting which words are linked and which are not. Second, this step makes the process of displaying each page easier to do, because the display routines can check for the presence or absence of the high bit, and then select the appropriate display colors accordingly.

The only task that remains to be solved is the step of unlinking a previously linked phrase. The solution is simple. When the user requests a link for a given word, the system first determines if the word has already been linked. It makes this determination by comparing the ordinal value of the character that is currently loaded beneath the cursor (at the cursor insert position). If the value of the current character is greater than *$80*, then the character must already be linked. In this case, the process just described is reversed — the high bit of each character in the phrase is cleared, returning it to its original ASCII value. Last of all, the hot-link token that was previously appended to the end of the phrase is removed. Voilà back to normal!

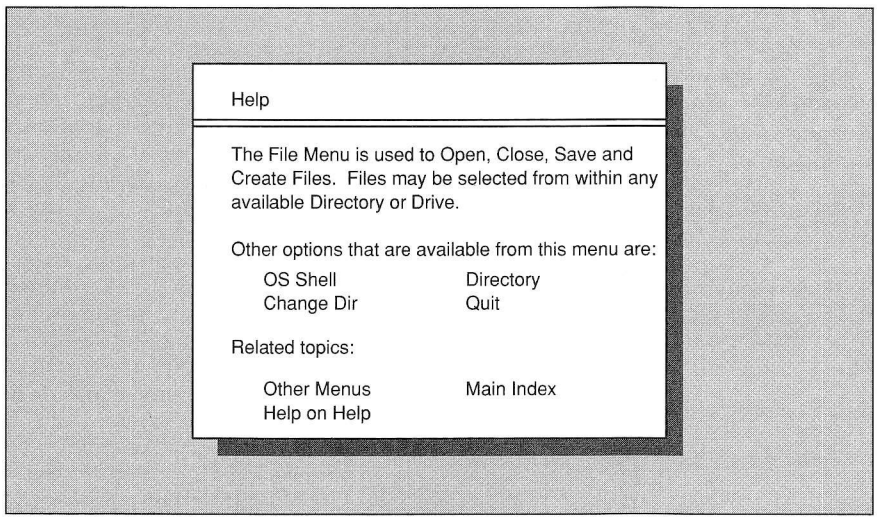


Figure 2: *Sample hypertext screen*

```
Const   MaxLinesPerPage = 15;
    MaxLineWidth    = 60;
Type    Help_Record = Record
        HelpLines : Array[1..MaxLinesPerPage] of String[100]; {Leave room
                                                               for links}
    end;
```

Example 2: *Defining the* Help_Record *data structure*

```
If Cursor is on a valid alpha character ([Succ(' ').. '} ']) then
   Begin
        Start := Find 1st char in the phrase;
        Stop  := Find last char in the phrase;
        For CurrentChar := Start to Stop do
            Set Char[CurrentChar] High-Bit;
        Attach link
   end
```

Figure 3: *Pseudocode description of the algorithm for placing a link*

```
Helplines[8] = [' OS Shell   Main Index '] {As a string}

     =  [ $20,$20,$20,$20,          {As Hex}
        $4F, $53, $20, $53, $68, $6C, $6C]
```

Example 3: *Before linking, the characters on line 8 look like this*

The Hypertext Editor

The hyperdata file-editor routines appear in the HyprText unit (Listing One) lines 77 – 288. The editor provides common routines for horizontal and vertical cursor movement, backspace, and character/line deletion. It's nothing more than a simple, screen-oriented line editor that can deal with both embedded hot-link tokens and characters that have their high bits set.

In the process of dealing with an ASCII character that has the high bit set is very simple. Because the only data allowed during data input are valid ASCII characters, we can be sure (corrupted data files notwithstanding) that any character with an ASCII value greater than 127 is part of a hot-linked

(continued from page 26)
phrase. To properly display this data, all that is required is to reduce the decimal value of the characters in question by *$80*. The routine *Show_HelpLine*, and its ancillary routine *Write_Char*, handle that task.

Show_HelpLine is called by *Show_Help_Page* to display each line of text. *Show_HelpLine* performs two related tasks. First, it determines the color attribute with which each character is displayed. This determination gives the system the capability to show hot-linked items in a different color and/or intensity than the color or intensity of normal text. Second, *Show_HelpLine* calls *Write_Char*, which decodes each ASCII character and reduces the character's value by *$80* if necessary before displaying that character. That's all there is to it.

To understand how the editor deals with embedded hot-links, consider how a line editor works. Given a line of ASCII text, the editing routine enters a loop. The user is permitted to enter valid characters, which are inserted into the line at the current cursor position. Editing commands are provided to allow the user to quickly alter the input data via such steps as deleting all of the characters from the cursor to the end of the line. The routine normally exits when the user presses a valid terminating keystroke, which is often just a press of the Return key.

The editor works in the same way. If a valid character is entered by the user, the character is inserted into the line at the cursor position, or at least at what appears to the user to be the cursor position. Remember that each line may contain hot-link tokens, which consists of data that are not actually seen by the user. What appears to the user to be column 10 for example, might actually be character 16 or character 20 once hot-link tokens are taken into account.

Consider Example 5. Let's presume that the cursor is located in column 17, just above the letter "M" in the word "Main Index." Also, presume that no hot-links are present in the line. In this case, the actual character number in the string is the same as the column number that appears to the user. In our example, if the user presses any valid character, that character is inserted into the line at column 17. Now let's say that the user previously linked the phrase "OS Shell," and that the cursor again is located above the letter "M," as shown in Example 6.

Just as before, the cursor appears to the user to be located in column 17, but the editor knows better. The actual character number is 19; the difference is due to the fact that a 2-byte hot-link token has been inserted into the line after the phrase "OS Shell." If the user presses any valid character at this point, then the character is inserted into the string as character 19. From the user's viewpoint, the result is identical to the process just described for the case where no embedded hot-links are present.

Two routines worth mentioning are *Determine_Actual_Line_Pos* and *Link_Count*. *Determine_Actual_Line_Pos* converts the visual column number that appears to the user (and is tracked by the editor) to the actual column position just described. *Link_Count* returns the number of hot-links found on a given line. This routine works by counting the number of null characters — our hot-link token delimiters — in a line. All of the work required to handle the process of editing text that contains embedded hot-links can be performed by these two routines, making the inclusion of additional editing routines much easier.

The editor itself is called and managed by the routine *Help_Editor*, which provides the necessary setup, file-handling, and page-manipulation routines required by the editor itself. *Help_Editor* also provides a simple screen display and an editing-command help window for the user — the latter is an excellent place for you to add your user-interface touches.

The discussion of the design considerations of the editor brings up an important point not only with respect to our hypertext system, but to user interface design in general. It doesn't make a bit of difference what is going on beneath the surface, as long as that activity does not interfere with the user. Case in point: Compare word processing on the Macintosh with word processing on the PC. On the Mac, the user never explicitly inserts font controls into a document. Instead, the user just selects the font they want from a menu and the rest is handled by the Mac beneath the surface. On the PC, the user practically has to learn a foreign language in order to instruct a word processor to change fonts mid-paragraph.

```
=  [ $20,$20,$20,$20,
        $CF, $D3, $A0, $D3,
        $E8, $EC, $EC, $00,$05]
```

Example 4: *After linking, the characters on line 8 look like this*

```
Helplines[8] = [' OS Shell  Main Index '];
                                ^
                          (Character 17)
```

Example 5: *The actual character number in the string is the same as the column number that the user sees*

```
HelpLines[8] = [' OS Shell<xx>  Main Index '];
                 ^     ^
            (Hot link)   (Character 19)
```

Example 6: *The phrase "OS Shell" was previously linked; cursor is on the letter "M"*

The Hypertext Engine

A hypertext engine is a collection of routines that permit a user to access the data contained in hyperdata files. Functionally, a hypertext engine is equivalent to a random-access file browser; it permits the user to scan records (or pages, in this case) in any order. The only difference with respect to a hypertext engine is that the user never directly specifies a record/page number. Instead, the user moves from one page to another by the selection of hot-links; the hot-links contain the new page numbers. The page number makes no difference whatsoever to the user — the user need be concerned only with the context of the hot-link keywords and the user's own interest in pursuing a given contextual flow.

The hypertext engine itself is shown in Listing One in lines 290 – 319 and 367 – 548. Listed first are the I/O routines that read and write the hyperdata files created and used by the system. Because these routines operate on simple binary files, and because binary file I/O is both easily understood and well-documented elsewhere, a discussion of these routines is omitted here.

The main task performed by the hypertext engine is to provide a user interface that permits the user to navigate hyperdata files. The engine creates the illusion of a highlighted selection bar for the user. A related task performed by the hypertext engine is to track the user's navigation of the hyperdata file. The process of tracking is required in order to allow the user to explore any contextual path to and then back out of that path to the original starting point. The engine handles this process by maintaining a local stack that tracks both the hypertext pages that the user visits and the order of the user's visit to the pages. (The *Stack* data structure is shown in Listing One, lines 438 – 442 and 444.)

The technique of tracking the user's navigation around the system is not

(continued on page 31)

(continued from page 28)
very difficult. Once the user selects a hot-link, the page number contained in the hot-link is pushed onto the local stack. If the user decides to back out of a path by pressing the PgUp key, then each preceding page number is popped off the local stack (which, in effect, reverses the user's path). This process continues until the root has been found, the user chooses to move down another path, or the stack is empty.

In addition to engine's ability to track the user's navigation around the system, another very useful feature is provided by the engine as a result of its local stack. The programmer can specify a starting page and a home page, thereby priming the local stack. This feature allows you to jump into the hyperdata file at any point and still permit your user to return to a main index or a home page. For example, let's say that you have written a database application and you've used this hypertext system as the basis of your pop-up help system. Further, let's say that the user has requested help on sorting records. As the programmer, you want the help system to first show the page concerned with sorting. This step is accomplished by passing a valid page number to the hypertext engine to be used as the starting page on the topic of sorting. A home page, usually used for a main index, can also be specified as a startup parameter for the engine.

One last point of interest is the method used to find embedded hot-links on a given page. The technique used for embedding hot-links directly into the data for each page has already been discussed. But because the system does not keep master hot-link tables, the links must be decoded at run time. Once a page has been loaded into memory and displayed, the recursive routines *Find_Next_Link* and *Find_Prev_Link* handle the decoding for you. *Find_Next_Link* accepts a line and column number for the current page, and then searches the page from that coordinate to the end, looking for an embedded link. *Find_Prev_Link* works in the same manner, except that it starts at a given coordinate and searches backwards toward the top of the page. The two routines allow you to start at any point on the screen, and finding the next or the previous hot-link. Once the new hot-link is found, it's highlighted by the selection bar, and can be invoked by a press of the Return key. Pages without hot-links terminate the current contextual path.

The Supporting Cast

The other routines in Listing One (see lines 32 – 75) are purely to provide support. The *Make_String* procedure creates a string of a given length from any ASCII character. *Draw_Box* draws single or double-lined boxes using ASCII line-drawing characters and the coordinates provided as parameters. (All simple stuff.) *Read_KeyBoard* accepts user input, one character at a time, and detects when the user presses the Ctrl and Alt keys.

Using the HyprText Unit

The code in Example 7 shows how easy it is to use the HyprText unit in your own programs. Line 9 shows a call to the editor, *Help_Editor*, that passes the name of the hyperdata file to be used. Once the user is finished editing the file, the program calls the procedure *Do_Help* on line 10, and passes to this procedure the name of the hyperdata file to be used. The other parameters passed to *Do_Help* specify the starting and home

(continued from page 31)
page numbers used to prime the stack (as described earlier).

If you want to use this system for pop-up help inside your own programs, you may want to add routines to save the text screen before calling *Do_Help*, and to restore the text screen once *Do_Help* terminates. Also, to keep the length of the listing to a minimum, I omitted most of the error-checking steps that a commercial application should have. You may want to add more error-checking routines to meet your own needs.

The Epilog

Well, there you have it — a hypertext system that is simple to program and simple to use. I hope that this system offers a useful addition to your programming toolbox.

I am sure that you can think of additional uses and enhancements to this system. What about converting this system to a graphics-oriented environment, such as the BGI? What about adding sound hot-links? Or adding a utility that reads text files, searches for keywords, builds an index, and then automatically builds hyperdata files? You could also add compression routines to reduce the storage requirements of hyperdata files.

Nobody ever said that programming the PC was supposed to be dull!

```
 1 Program HelpTest;
 2 Uses Crt,HyprHelp;
 3 Var FileName : String[80];
 4 Begin
 5  ClrScr;
 6  {If you want to save the current text screen, do it here.}
 7  Write(TEnter the file name: U);
 8  Readln(FileName);
 9  Help_Editor( FileName );
10  Do_Help ( FileName,1,1);
11  { Restore the text screen here, if you saved it before }
12 end.
```

Example 7: *Code to incorporate the HyprText unit in other programs*

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listing begins on page 86.)

Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

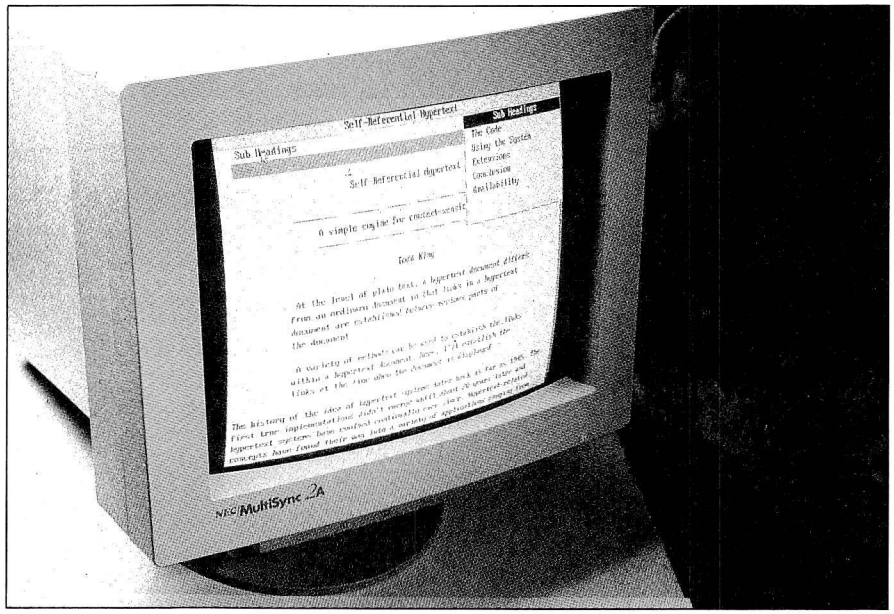
A Self-Referential *Hypertext Engine*

A simple engine for context-sensitive help

Todd King

The history of the idea of hypertext systems dates back as far as 1945. The first true implementations didn't emerge until about 20 years later and hypertext systems have evolved continually ever since. Hypertext-related concepts have found their way into a variety of applications ranging from context-sensitive help to document-management systems to user-programmable systems such as Apple's HyperCard. Even multimedia hypertext systems (which are the purest form of hypertext) are now within our reach.

In this article, I'll discuss how you can build a hypertext system that can be used for the development of a context-sensitive help system or for the display of hypertext documents. The type of hypertext system presented is a "text-based" (no graphics or sound) system. This type of system is sufficient for many applications, and is easy to implement.



At the level of plain text, a hypertext document differs from an ordinary document in that links in a hypertext document are established between various parts of the document. These links allow you to reposition yourself within the document by navigating along the links. In some ways, a link is an iconic bookmark to which you can move at any time. In the case of a graphical or multimedia hypertext system, other actions may also be associated with a link. For example, a link may display a particular photograph, or even produce music.

Just as a variety of hypertext systems exist, a variety of methods for establishing the links within a hypertext document also exist. In the application presented here, I establish the links at the time when the document is displayed. This step is accomplished as follows: Each topic in the hypertext document has a phrase associated with it. This phrase serves as a tag which, if referenced in another topic, automatically establishes a link. This type of approach allows you to build up your hypertext document either by adding topics when you need them or when you find that certain words or phrases require elucidation. Because the new topics will automatically be linked to all other topics that reference them, you don't have to spend time managing the links — all you do is write the document.

Todd is a programmer/analyst with the Institute of Geophysics and Planetary Physics at UCLA where he works on various software projects. He is writing a book, Dynamic Data Structures, *which will be published by Academic Press, late in 1990. Todd can be reached at 1104 N. Orchard, Burbank, CA 91506.*

The Code

The code for the hypertext document display program (which I call *hyper_d*) is presented in Listing One, page 92. *hyper_d* is invoked without any command-line arguments (for the sake of simplicity). When the program is invoked it first opens the document file called "HYPER.TXT." *hyper_d* then calls

Dr. Dobb's JOURNAL
SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

PUBLISHER *Peter Hutchinson*

EDITORIAL
EDITOR-IN-CHIEF *Jonathan Erickson*
MANAGING EDITOR *Monica E. Berg*
TECHNICAL EDITORS *Michael Floyd, Ray Valdes*
ASSOCIATE MANAGING EDITOR *Janna Custer*
CONTRIBUTING EDITORS *Al Stevens, Jeff Duntemann, Martin Tracy, David Betz, Tom Genereaux, Andrew Schulman*
COPY EDITORS *Rhoda Simmons, Pamela Dillehay, Nan Fornal*
EDITOR-AT-LARGE *Michael Swaine*

ART/PRODUCTION
ART/PRODUCTION DIRECTOR *Larry L. Clay*
ART DIRECTOR *Michael Hollister*
PRODUCTION SUPERVISOR *Amy Shulman Lesovoy*
TYPOGRAPHERS *Teresa Raines, Margaret Anderson, Charlene Carpentier*
COVER PHOTOGRAPHER *Michael Carr*

CIRCULATION
DIRECTOR OF CIRCULATION *Maureen Kaminski*
CIRCULATION MANAGER *Randy Robertson*
CIRCULATION PLANNING MANAGER *Manny Sawit*
DIRECT MARKETING MANAGER *Andrea Weingart*
NEWSSTAND MANAGER *Sarah Forsman*
DIRECT MARKETING COORDINATOR *Susan Bauman*
PROMOTION COORDINATOR *Pam Moore*
FULFILLMENT COORDINATOR *Anne Jean*

ADMINISTRATION
VICE PRESIDENT OF FINANCE *Kate Deschamps*
CONTROLLER *Mary Collopy*
CREDIT MANAGER *Betty Arsene*
ACCOUNTING SUPERVISOR *Renate Kernke*
ACCOUNTS RECEIVABLE *Wendy Ho*
ACCOUNTS PAYABLE *LuAnn Rocklewitz*

MARKETING/ADVERTISING
DIRECTOR OF SALES AND MARKETING *Karla Spormann*
ADVERTISING COORDINATOR *Laura Stack Pullen*
MARKETING ASSISTANT *Sara Noah Ruddy*
ACCOUNT MANAGERS *see page 160*

M&T PUBLISHING INC.
CHAIRMAN OF THE BOARD *Otmar Weber*
DIRECTOR *C. F. von Quadt*
PRESIDENT *Laird Foshay*
VICE PRESIDENT OF PUBLISHING *William P. Howard*
VICE PRESIDENT/GROUP PUBLISHER *Randall L. Stickrod*

DR. DOBB'S JOURNAL (USPS 307690) is published monthly, except semimonthly in December, by M&T Publishing, Inc., 501 Galveston Dr., Redwood City, CA 94063; 415-366-3600. Second-class postage paid at Redwood City and at additional entry points.

ARTICLE SUBMISSIONS: Send manuscripts and disk (with article and listings) to the editorial assistant 415-366-3600.

***DDJ* ON COMPUSERVE:** Type GO DDJ.

***DDJ* LISTING SERVICE:** 603-882-1599. Supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Type *listings* (use lowercase) at the login prompt.

SUBSCRIPTION: $29.97 for 1 year; $56.97 for 2 years. Foreign orders must be prepaid, including the additional postage (air or surface) in U.S. funds drawn on a U.S. bank. Add $13 per year for surface mail; add $36 for airmail to Canada and Mexico; or $26 for airlift to all other countries.

POSTMASTER: Send address changes to *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. **ISSN 1044-789X**

CUSTOMER SERVICE: For subscription questions, call toll-free 800-456-1215. For subscription orders or change of address call toll-free 800-456-1215 (U.S. and Canada) or write *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. For book/software orders call 800-533-4372 (in California 800-356-2002).

FOREIGN NEWSSTAND DISTRIBUTOR: Worldwide Media Service Inc., 115 E. 23rd St., New York, New York 10010; 212-420-0588 FAX 212-420-1265.

Entire contents copyright ©1990 by M&T Publishing, Inc., unless otherwise noted on specific articles. All rights reserved.

The Audit Bureau

(continued from page 34)

the function *build_hyperbase( )*. This function builds an index of topic tags for all of the topics in the file, and stores the location of the beginning of each topic within the file. This tag data base, called "Hyperbase," is used by the display and navigation routines to establish links and to move within the document. The definition for the Hyperbase structure, as well as other definitions, can be found in Listing Two, page 94.

A variety of methods can be used to establish the links within a hypertext document

The next function called by *hyper_d* is *enter_text( )*. This function references a given topic by its tag, vectors to that topic in the document, and then allows navigation within the document. *enter_text( )* returns each time a tag is selected within a document, and then evaluates to the index in Hyperbase of the selected tag. As long as the Esc key is not pressed (which returns a special tag ID of -1), *enter_text( )* is called with the tag that represents the current selection. This step repositions you in the document and begins the entire cycle again. If the Esc key is pressed, the application performs some cleanup operations and then exits.

enter_text( ) could also have been implemented as a recursive call, rather than placing it in a *while( )* loop. This function was placed in a *while( )* loop because this approach minimizes the demands for stack space. Imagine the results of making each topic a separate function, or calling *enter_text( )* as the navigational step rather than using looping. You are free to navigate through the document and jump along links as many times as you like, so at some point a recursive implementation would exceed the limits of the application's stack.

Several other functions and structures are used in the application. Rather than describe each one here, I created a hypertext document that describes the entire application. The text for this document is contained in Listing Three, page 94. The structure of the hypertext documents used by this system is very similar to the structure of a conventional document. The listing is quite readable without using *hyper_d*, but the use of *hyper_d* makes the process of reading the listing more efficient and more fun.

Using the System

When you've created a hypertext document that you'd like to view, simply place the document in a file called "HYPER.TXT" and run *hyper_d*. Provided that your document is structured properly, you will be placed at the first item in the first topic, and then set free to navigate through the document. Each link to another topic is displayed as bold type on monochrome screens, and as yellow type on color screens. The currently selected item is displayed in inverse video. To move the highlight around within the document, press either the left or the right cursor key. This step will move you either backward one item (left cursor) or forward one item (right cursor). Once you've highlighted a tag of a topic to which you'd like to move, press the Enter key.

To create a new document, you have to follow some simple rules so that *hyper_d* can locate each topic. A topic begins with a line that consists of a form feed that is followed by a new line. The next line that follows is the tag string for the topic. Everything that follows the tag line is text for the topic. You can insert any characters into the text portion of a topic, including the graphic characters in the PC's character set. Graphic characters are handy because they give the appearance of being graphically based, when in fact they're only character based.

The way in which you embed a form feed into your document differs from editor to editor. Here's how to do it using a few of the editors that I use regularly. In Turbo C, press Ctrl-P and then press Ctrl-L. In vi, press Ctrl-L while the program is in insert mode. In Microsoft Word, insert a page break by pressing Shift-Ctrl-Return.

Extensions

Many obvious improvements could be made to *hyper_d*. Only a minimal set of functions has been implemented in order to present the elements of a hypertext system in the clearest possible way. One obvious improvement would be to add support for a full range of navigational keystrokes, such as the up and down cursor motions. Another improvement would be to "compile" the text, rather than take the "interpreter" approach that was done in this article. This technique would involve the process of calculating the links (as performed by *build_hyperbase( )*) and then recording the calculation in some way

(continued from page 36)
so that navigation can begin immediately when the document is called.

Some features of more powerful hypertext systems include navigation history lists, which allow you to retrace your steps through a document; the possibility of integrated text and graphics; and scrollable windows. Even though these features (and many more) are expected of hypertext systems today, applications for text-only hypertext still exist. The hypertext system for the documentation of the source code presented in this article is one example.

Conclusion

I think Apple will be remembered fondly for bringing hypertext systems into the hands of the average person — not so much because the company developed HyperCard, but because Apple gave it away for free, bundled with every Mac. Even though a PC is just as capable of running hypertext systems as a Mac, there isn't a single (as far as I know) PC-based hypertext development system that is given away free or almost free. The PC-based application presented in this article is available for anyone to use, free of charge. You are also welcome to use it as a seed for the development of a more fully featured display system. If you do expand this code directly, please keep the results in the public domain.

This code was developed and tested using Turbo C 2.0. It also uses certain library functions that are unique to Turbo C and are related to screen output. These functions are: *clrscr( )*, *cputs( )*, *gotoxy( )*, *textbackground( )*, and *textcolor( )*.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

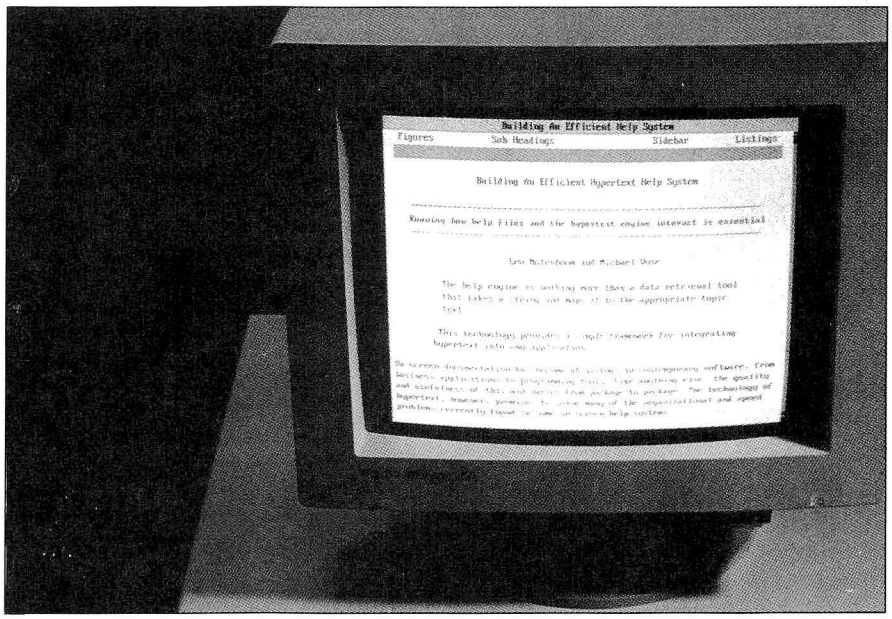
(Listings begin on page 92.)

Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

Building an Efficient *Help System*

Knowing how help files and the hypertext engine interact is essential

Leo Notenboom and Michael Vose



On-screen documentation has become ubiquitous in contemporary software, from business applications to programming tools. Like anything else, the quality and usefulness of this aid varies from package to package. The technology of hypertext, however, promises to solve many of the organizational and speed problems currently found in some on-screen help systems.

On-screen documentation is useless unless it makes finding information easy, provides that information quickly, and delivers all the information users need. Some kinds of information — command references, API references, example code, and the like — lend themselves more readily to on-screen help than do others. Hypertext can equip a help-system programmer with the tools needed to create on-screen documentation that is fast, easy to navigate (even through voluminous text), and comprehensive. Hypertext isn't magic; it's actually quite a simple idea. In a help system, it's the association of a programmed action with an area of viewed text.

This article explains how help files and the hypertext help engine of a typical hypertext-based on-screen help system fit together, using the Microsoft Advisor as the example. With an understanding of how this technology works, you can use it and associated tools to add on-screen documentation to your libraries and programs or access on-screen help provided by any other program that uses similar technologies.

Leo, a software development engineer and manager for Microsoft, is the designer of the Microsoft Advisor. Michael is a coeditor of OS/2 Report *newsletter.*

Hypertext in Context

As already defined, hypertext is simply the association of an action with an area on the screen. Thus a "button," which is just a visible region on a screen, might have associated with it the action of looking up help text on a predefined string of characters. When a user selects the button, the associated help text is located and displayed.

In practice, this movement between help screens via buttons or an action such as pressing a help key is generally referred to as a link or cross-reference. There are two primary types of links:

• **Implicit Links** When a user selects a word on the screen and requests help by pressing F1 or clicking a mouse, the application looks up help on that word and displays the resulting help text. Within the help text itself, a user can place the cursor on any word within

(continued from page 40)
the help text and press F1 again to get further help. This second lookup operation is made possible by implicit links.

• **Explicit Links** Encoded by the help file author, explicit links specify a region within the text (often represented by a button) and the help text to which that region refers. For example, a help screen may present an area labeled <Example Code> that was defined by the help author to be linked to the phrase *abs.example*. When the user selects that button, the application looks up the phrase *abs.example*, and displays the resulting help screen.

Explicit links come in two flavors — normal and local. Normal explicit links operate exactly as in the previous example. They differ from local explicit links in that they can reference different help files from the one actually containing the button. Local explicit links, on the other hand, are restricted to linking to help text present in the same help file as that containing the button. As we'll see later, local links have advantages and disadvantages.

To implement these hypertext links

Turning the Tables

The tables that connect the different parts of a help database are created by the HELPMAKE utility and are stored along with a header at the beginning of every help file (see Figure 3). Every help file has the following structure: A header that contains the identifier and version of the file, location of the subsequent tables, and some additional information; the topic map, an array of file positions of each of the topics in the help file, plus an extra entry to mark the end of the file; the context map, which matches a context number to a topic number; the context string table, which defines the strings when help exists and determines their context numbers; optional keyword compression table, which contains the keywords removed from the topic text during compression; an optional Huffman compression table, which contains the decode tree created during compression; and topic text.

Topic text itself contains more than just the displayed text — it also carries control, color, and cross-reference data. The control information includes a number representing the size of the uncompressed topic in bytes. Each subsequent line of text consists of the ASCII text to be displayed; attribute/color information, such as bold or underline, that an application can then map to colors if desired; and explicit link information, which specifies the areas within the line that are hot spots, and the string local context number with which the explicit links cross-reference.

When a help file is compiled, the HELPMAKE utility makes three compression passes on the topic text. Run-length, keyword, and Huffman compression are each applied to the text in turn.

Run-length encoding is the replacement of runs of characters (three or more of the same character) with a special token signifying the run, the character that is to be repeated, and the number of times it is to be repeated. Encoded this way, any run of 4 to 255 characters can be replaced by 3.

Keyword encoding is the replacement of commonly occurring words or phrases with a 2-byte token. The text is analyzed for the words occurring most frequently, and the most frequent words are collected into a key-phrase table. Occurrences of these words in the text are then replaced by a token whose value is an index into this key-phrase table. For example the word "the" is used frequently in normal English text. If you remove this word and replace it with a 2-byte token, the savings of 1 byte per occurrence adds up quickly.

Huffman encoding is simply a bit-for-byte replacement. The text is again analyzed for frequently occurring bytes, and the most frequent bytes are replaced with a shorter bit pattern. For example, if you can replace 1000 8-bit bytes with a 2-bit pattern, you save 6000 bits, or 750 bytes. To restore a Huffman-compressed file requires a table that a decompression routine can use to restore the 2-bit pattern to its original 8-bit value. The side effect of Huffman compression is that less frequently occurring bytes often get replaced by a bit pattern longer than 8 bits. However, the net gain due to the frequency of the smaller bit patterns quickly outweighs the loss due to the growth of the longer ones.

Microsoft offers a "Programmer's Workbench Toolkit" that contains the Advisor API Library and Documentation. Call 1-800-426-9400 to obtain details. — L.N., M.V.

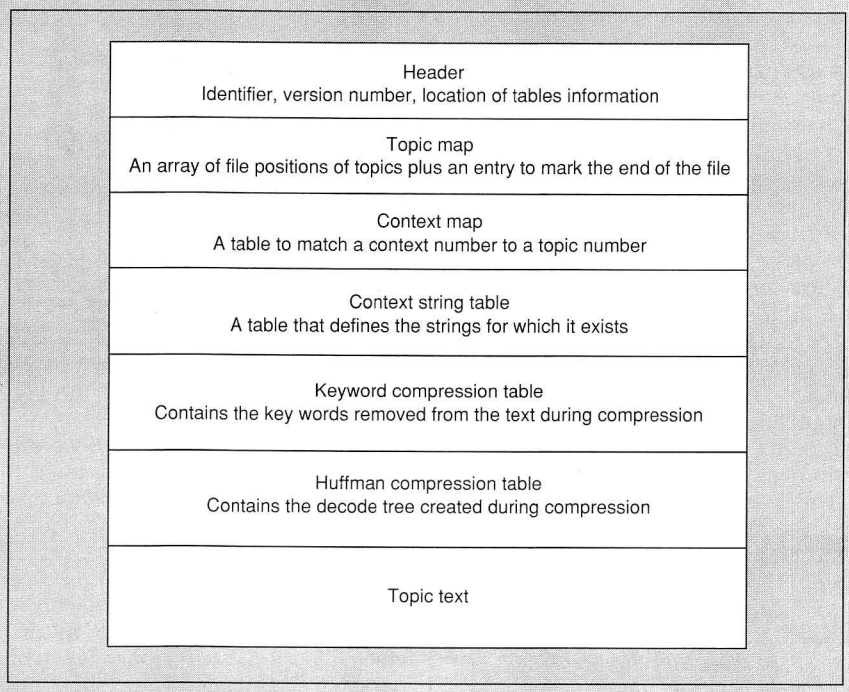


Figure 3: *The components of a help database*

(continued from page 42)
to assist users in navigating help text, a help-system author can benefit from understanding the data structure known as a help database.

Help Files and Databases

To make on-line help fast requires a way to quickly map individual words or phrases to some textual or graphic information. Associating the explanatory text or graphic information with more than one word or phrase may also be necessary. The help file contains the information that allows help retrieval to be both fast and flexible. A help file comprises one or more help databases. A help database contains the words that help is available for and the information associated with those words.

The words and phrases for which help can be requested, called "context strings," and their associated help text, referred to as "topic text," are authored using a text editor or word processor and are compiled into help files using the HELPMAKE utility. HELPMAKE creates the links between the context strings and topic text and also compresses the topic text. HELPMAKE can also decompress, or "uncompile," an entire help file into its original, editable form. (See the accompanying text box, "Turning the Tables" for information about the structure of a help database and the techniques used to compress it.)

During help-file compilation, the actual connections between context strings and topic text are made through a series of three lookup tables used by the Advisor when a help request is made (see Figure 1). The context strings supported by the help file are stored as a simple array in the first table, and the help text contained in the file can also be viewed as an array of topics. The job at lookup time, then, is to map the given context string to the correct piece of topic text to which it refers.

The first table, the context-string table, is just a simple array of context strings. The index of a string in this table becomes its "context number." The second table maps the context number to its corresponding topic. The entry indexed by the context number in this "context map" contains the corresponding "topic number." The last table maps the topic number to the actual position within the help file that contains the topic text. From this last table, the compressed size of the topic can be calculated (the difference between two successive file offsets; this number can then be used for memory allocation) along with the predicted end of the help file (the file offset of the last-plus-one help topic used for concatenated help files, described later).

The Advisor optimizes this lookup process by loading the mapping tables (and the decompression tables described in the text box) into memory once, wherever possible, and leaving them there until the application instructs it to discard them.

Because a context string is transformed into a number representing its position in the context-string table, an application can discard this string once the transformation has been made. In fact, this context number is returned to the application in a form that also identifies the specific help database to which it applies when more than one help database is being used. This context number can be used to map backward if necessary to determine what string was originally looked up and what database it came from.

One special type of context number provided by the Advisor for the application is called a "local context number," or simply "local context." Local contexts are not associated with a con-

(continued on page 47)

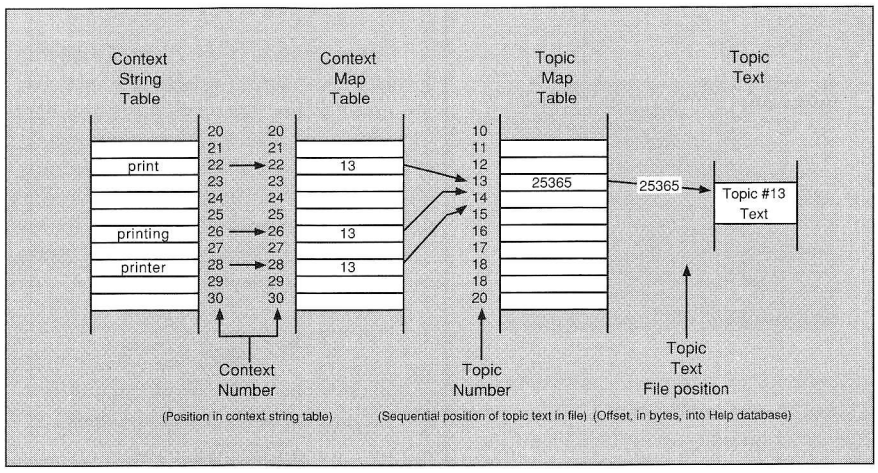


Figure 1: *The look-up tables for mapping context strings to topic text. Note that three related context strings can map to the same topic text*

(continued from page 44)
text string; rather, they are directly encoded with the desired topic number.

Local contexts are identified by the help-file author at the time the file is written and are compiled specially by the HELPMAKE utility. Explicit links often use local contexts instead of normal contexts because the 2-byte topic number with which a local context is encoded is much smaller than the string normally coded into an explicit link. The only restriction on local contexts is that explicit links that reference local contexts must have the topic text associated with each local context present in the same help database.

Multiple help databases can be concatenated to form a single help file for a given application. When a help file is opened, the hypertext system, using the three tables previously mentioned, examines what should be the end of the database in that file. If instead it finds the beginning of another help database, it opens that as well and repeats the end-of-database examination. By repeating this operation, multiple help data files can be combined simply by using the MS-DOS Copy command, and even applications that support only one help file can be "fooled" into using several help files disguised as one.

The Help Engine

The help engine is nothing more than a data retrieval tool. The engine takes a string and maps it to the appropriate topic text. It simply searches the data structure that is a help file and makes the desired information available to an application.

The Advisor help engine knows very little about its environment — except for the OS/2 version, which does make some assumptions about memory management and file I/O. The MS-DOS version of the help engine relies on the application to handle memory management and file I/O.

The engine enforces the help-file format and its data structures but leaves determinations about how to use retrieved data to an application. The help engine has to be told what help files to deal with, and an application can specify multiple help files.

In addition to routines to query for and retrieve help text, the help engine provides routines that the application must use to perform decompression after the topic text has been located. The help engine also has routines that an application can query to discover if a topic has cross-references or to read only the color or control information in a topic.

The help engine uses application-provided handles for memory management and for file I/O and can therefore deal with many different memory and I/O schemes.

The Application's Job

The application program that uses the help system must perform several important functions of its own (see Figure 2), including providing the user interface to the help system and supplying the interface to its environment.

The most important of these functions is the user interface for displaying help information and interacting with the user and the help screens. The application defines what text gets parsed into a context string when a help request is made, and it then passes that string to the help engine, which mounts the search for that string. For example, if the cursor resides on a menu or an open dialog box, the application's parser must be able to decide if help is appropriate for that object. The application

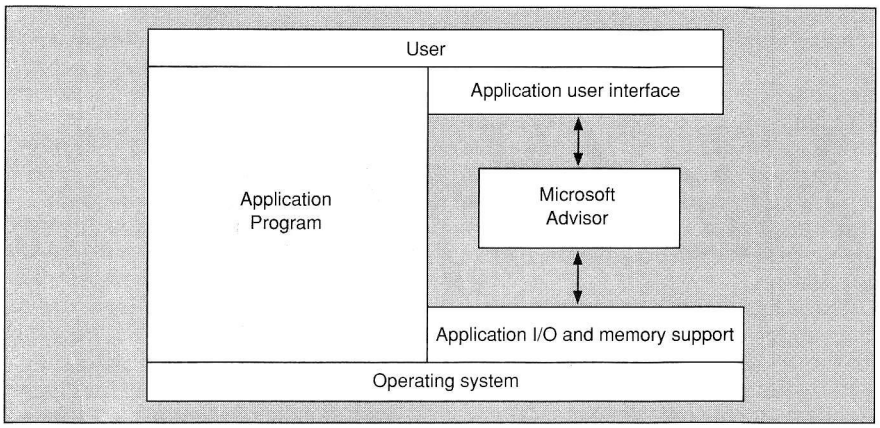


Figure 2: *The help engine's place within the hierarchy of an application*

also interprets all control and cross-reference commands, as well as handling multiple help files. For example, if an application has five open help files and a user asks for help on a string, the application must use the Advisor to query each help file in turn to find the desired string match.

The application must also handle failures and be able to display a message or beep the speaker when no help is available. Conversely, the application must determine what action to take if there is help on the same context string in more than one help database.

In addition, the application must process control information. The Advisor offers a history function that lets a user move backward through the 20 most recently viewed help screens. Because the context numbers returned by the hypertext system uniquely identify each help database and context string, the history function simply keeps a circular queue of the 20 most recently accessed context numbers. The application must provide a keystroke or clickable screen button to engage this history function. The application parses a history request and then calls the help engine's history function to retrieve and display previous help screens.

Under MS-DOS, the application must also provide the interface between the Advisor and its environment, including memory management and all direct file I/O support.

Taking Advantage

The hypertext technology described here can be leveraged in two ways: By supplying supplemental help files for use by applications using the hypertext system and by incorporating the system into new applications. Supplemental help files offer the ability to customize and expand on-line help to fill a variety of needs. New applications being written can benefit from the enforced consistency in file format in that all help files can be used by any application that also uses the Advisor. This technology also provides a simple framework for implementing hypertext in any application and provides a painless way to compress text without any significant access-speed penalties.

Searching for information was the primary application of computers from the beginning. Today's hypertext-based on-screen help systems can begin to make the information that people need the most more readily accessible than ever before.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 4.**

C++ File Objects

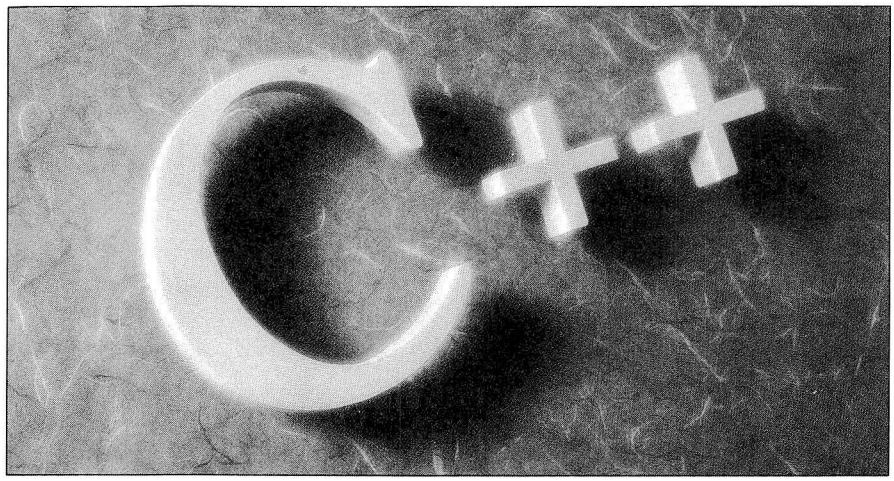
Writing a base class is the key to programming with an object-oriented language

Kevin Weeks

A few months ago I wrote a database program for a client and found myself manipulating multiple files in multiple directories on multiple drives using multiple devices. Although I'd run into this sort of challenge many times before, this particular program involved some variations on the theme that didn't quite fit my previous work. Nevertheless, I went ahead and wrote the program in C, rewriting file access/handling routines I'd developed before.

As an afterthought, I decided to use C++ to design a basic object or objects that would at least minimize some of these problems in the future. Although there are a number of examples of file-type objects around, I haven't run across any that are capable of taking a partial file specification, such as TEMPFILE.DAT, and completing it by adding the drive and path components. Failure to do so can result in unpredictable references. For instance, A:TEMPFILE.DAT obviously refers to a file on drive A in the current subdirectory. But what is the current subdirectory? Is it what the user intended? I came to the conclusion that a file object starts too high up the tree. The first step is creating the file specification, the second is handling the file.

Kevin had been coding in C for six years. He is working for Computational Systems, a vibration analysis company in Knoxville, Tenn. Kevin can be reached at 508 Valparaiso Rd., Oak Ridge, TN 37830.



Creating a good base class is like sculpting, start with a block of marble and then chip away everything that doesn't look like a statue. This took a while but the result was class *File_Spec*.

File_Spec

Although the code that defines the *File_Spec* class is large, it contains nothing that isn't specific to the creation and maintenance of a DOS/Unix-style file specification. Each component can be retrieved or modified, an incomplete file specification can be completed, and the object's current status is maintained and made available to clients. Of course, not all the methods have to be included in a given application. I have provided a single *File_Spec* module (FILESPEC.CPP, Listing Two, page 96) for simplicity's sake. If the individual methods are compiled separately and placed in a library, however, then only those methods actually used by a program will be included.

If you look at Listing One (FILESPEC.HPP, page 96) you'll see first that I named the fields (attributes in OOP-speak) *device*, *prefix*, *name*, and *suffix*. The idea behind these labels is to provide some degree of abstraction in case the object is ported to a system where the device doesn't have to be a disk drive, or the prefix a DOS/Unix-style path. (For instance, I've recently been coding on a Prime Series 50 machine where the device names are logical drives enclosed in <...>.) Next you should notice that the *device*, *name*, and *suffix* are implemented as arrays.

C++ books are big on the new operator and seem to use it at the drop of a *class hat*. I've gotten burned by poor *malloc( )* implementations a time or two, where memory ended up so fragmented that *malloc( )* lost all con-

(continued from page 50)
ception of order. Because there is no specification for memory management in C++, it seems wise to limit the potential for such problems when objects are potentially volatile.

device, *name*, and *suffix* are all relatively short; any waste space is minimized, reducing both the code and the time required to change them with the *change_XXXX( )* methods. The exceptions are *prefix* (the path) and *request*. Because these may vary in length from one character to as many as 78 characters, it seemed more reasonable to swallow the overhead and reduce RAM requirements in their case. If you're wondering what the *request* attribute accomplishes, it forces me to look and not touch.

Creating a good base class is like sculpting, start with a block of marble and then chip away everything that doesn't look like a statue

All strings that have to be returned are returned as a pointer to *request*, which is allocated as needed. Obviously I could reduce code and execution overhead even more if, instead of copying the *device*, *name*, and so on to another string and returning it, I simply returned pointers to the respective attributes. I must confess, however, that I might be tempted at some point to modify the attributes directly by using an alias, instead of playing by the rules. What can I say? I'm weak. Because I can't depend on self-discipline, I'm better off returning something that can't modify the originals. Besides, I have to create another string to return the complete file specification to clients anyway. Why not let the caller pass a string pointer for the method to fill? Two reason: First, without bounds checking there's no way to be sure the passed string is big enough, and second, this adds to the complexity of using the class methods, without really buying anything.

Making It

The first four methods in Listing Two are the constructors and the destructor. Constructors constitute a built-in setup

or initialization routine for an object. All you have to do is declare the object, and the compiler automatically calls the appropriate constructor. C++ also allows you to have more than one constructor (or method) with the same name (hence multiple constructors). Depending on the number and type of parameters passed, the compiler determines just which method you need. This is called "overloading" and is an example of polymorphism. More on this later.

There is a drawback to constructors: They always return a reference to the object created. If problems occurred, you'll have a malformed object. There's no such thing as an error or even a NULL return. Consequently, you should check the status (see *status( )* later) prior to using the object (see Class File for examples) and check *errno* for memory allocation errors following the object's creation. This necessity is irritating and unrefined, but we're stuck with it. On the positive side, because both of the objects presented here are aware of their own status, they won't misbehave if asked to do the impossible — they'll simply report an error back to the caller. Ideally, incorporating a reference to a separate Error class would enable you to use the objects without worries.

The first constructor simply creates an empty object that can later be filled by using the other class methods. The second constructor accepts a *reference* (pointer) to another *File_Spec* object and makes a copy of it. These two are pretty straightforward. The constructor that accepts a character string, however, isn't quite so obvious.

Following initialization of the various attributes, the (*char*string*) constructor uses a finite state machine to parse the string passed to it. This string may consist of any or all combinations of a *device*, *prefix* (path), *name*, and/or *suffix* (extension.) The *prefix* may be a full or partial path, but must end with a backslash to be properly recognized. Once the string has been parsed, checks are made both to see which components are missing and for valid file characters.

You indicate to C++ that a method is a destructor by prepending a tilde (~) to the class name. This message is automatically passed to an object whenever the object passes out of scope and is meant to perform any necessary cleanup. In the case of *File_Spec* this means deallocating any memory assigned to *prefix* or *request*.

The Rest of *File_Spec*

Status( ) returns the condition of the file spec. A non-zero return value indi-

cates a problem of some sort. Exactly what the problem is can be determined by examining the value. The low nibble of the low byte of the status word indicates whether or not the spec is complete; the high nibble of the low byte of the status word indicates invalid file characters. Next is, in my opinion, one of the most exciting aspect of object-oriented programming — operator overloading or polymorphism.

C allows you to create new types with the *typedef* operator. What it doesn't do is redefine operators to handle the new types. With C++, you can. This provides genuine language extensibility, which is one of the things that gets Forth programmers so excited (even if they can't figure out what they did a week later). Most of the C operators can be overloaded, and early versions of *File_Spec* got rather carried away with this capability. By dint of much pruning, we are left with the equal sign (=). In this incarnation, *File_Spec*= doesn't even do much, it just calls *copy( )*, a private method. Operator overloading is a powerful capability, and everyone will tell you it can be abused. They're right, but go ahead and abuse. Get it out of your system. I did. Now back to earth.

The *get_XXXX( )* and *change_XXXX ( )* methods should be readily understandable. Because both, *request* and *prefix*, have lengths associated with them, they are only reallocated if they aren't long enough. (I know. If I'm concerned enough about storage to make them dynamic to begin with, why don't I shorten them when possible. What can I say? GIGO is one thing, but someone has to pick up the garbage.) The change methods also keep the condition flags up to date. The *read_only( )* method is to prevent the change methods from working at inconvenient times, such as when the file is open. *Check_prefix( )* determines whether or not a path is complete. It does this by checking for a backslash as the first character. If there is no backslash, the path is relative to the current working directory. Next it looks for a period (.) followed by backslashes or another period. This refers either to the current directory or to the parent directory. The last step is to make sure the prefix ends with a backslash.

One of the principles of object-oriented programming is late binding. Without getting into details or specifics, this means that decisions concerning which functions to call or what parameters to pass can be delayed until run time, thus making a virtue of procrastination. The advantage is an increase in the flexibility of the program. In keeping with this principle, *File_Spec* objects do not attempt to automatically complete themselves until specifically requested to do so. This means that a partial *File_Spec* can be instantiated, passed around, copied, and modified without worrying about the validity or presence of a drive or path until those elements are necessary.

Complete( ) attempts to resolve an incomplete file specification. If a name was not specified or if any of the attributes have invalid characters, *complete( )* immediately returns a FALSE. If *complete( )* passes those tests, it checks for a device specification and, if one isn't found, calls DOS for the current drive using *ll_get_drive( )*. I wrote the assembly language module (LOWIO.ASM, Listing Five, page 111) to provide three functions. First, I needed a routine for getting just the current drive (without the current working directory). Second, I needed to be able to get the current working directory of another drive. Third, I wanted a routine to truncate a file at a position other than its beginning.

C allows you to create new types with the typedef *operator. What it doesn't do is redefine operators to handle the new types. With C++, you can*

The last step is to check for and, if necessary, complete the prefix. If the prefix is incomplete then *parse_prefix( )* is called. Space is allocated for a path name, and then DOS is called for the current working directory for the specified device via *ll_get_cwd( )*. Again, as in the constructor, a finite state machine is used to build the prefix. Relative references (. .\ and .\) are treated as DOS would. This routine, though, is a bit smarter, or takes a bit more for granted, than DOS. Multiple backslashes are ignored. For instance, \\ *subdir* \ is treated as \ *subdir* \ . More than two periods in a row are treated as two periods in a row: . . .\ *subdir* \ = . .\ *subdir* \ . And two or more periods in a row without a backslash are treated as two periods with a backslash: . . *subdir* = . .\ *subdir*. Obviously these fixes are not guaranteed to produce what you and/or your user had in mind, but the chance of possible harm seems minimal.

Class File

Class File (see FILE.HPP, Listing Three page 106 and FILE.CPP, Listing Four page 108) is the first derived class of *File_Spec*. Its purpose is to bundle the basic file operations such as *create*, *read*, *rename*, and so on into a single object. *File* is defined as a *:public* descendant of *File_Spec* so that all of the *File_Spec* methods are available to *File* and its clients.

The first File constructor is defined like this:

```
File::File(char *name, int open_flag
                = FALSE):(char *name)
```

The phrase *int open_flag = FALSE* is one of two points worthy of comment. This phrase defines a pass parameter, *open_flag*, and states that if none is provided to default to FALSE. Default parameters are sexy. They're a kind of "just give me the usual" to the compiler. The other point worth noting here is the phrase . . . *:(char *name)*. This is a call to the parent's class constructor. As you can see, the File class doesn't use the name parameter itself but instead passes it on to its parent, *File_Spec*. Although a child constructor does not automatically call its parent constructor, the parent destructor is automatically called.

Next, *exists( )* reports to its client on the existence of the file specified. In addition to reporting to clients, the *exists( )* method is used by the File methods *open( )* and *create( )*. With these two we come to a somewhat peculiar function call:

```
::close(tmp_handle);
```

C++ defines an "overload" directive that is intended to allow the programmer to use the same name for multiple functions and which, according to the books, should work like this:

```
overload    foo;
int         foo(char *);
int         foo(unsigned int);
```

Following this declaration, you should be able to issue calls to *foo(. . .)* and depending on the type and number of parameters passed, the compiler will select the appropriate function. Unfortunately, there is a bug in Zortech's 1.0 implementation and explicit overloading doesn't work properly. When I spoke to Zortech's tech support people about

this they recommended using the :: operator as a workaround. This operator is a scope qualifier (notice its use in all the method definitions to restrict their scope to their class). When used without a class name it is a global reference and enables me to access the standard C *close( )* function. I also use the :: operator to access the library versions of *open( ), read( ),* and *rename( ).*

As for *read( )* and *write( ),* under DOS, Unix, and many other operating systems, the file pointer is automatically advanced whenever a read or write operation is performed. While this is convenient in the case of a sequential file, most of the time it constitutes an undesired side effect (certainly when you're writing a random access database program). For this reason I added an auto-advance file attribute, which is set when the object is opened or created by ORing *F_ADVANCE* with the other file attributes. Consequently, if a File is opened or created with the *F_ADVANCE* flag it behaves just as you would expect it to. Without this flag, however, reads and writes don't lose their position in the file. Now just imagine a (descendant) database class using methods such as *retrieve( )* and *replace( )* instead of *lseek( ), read( ), lseek( ), write( ).* Seems just a hair more elegant, doesn't it.

Truncate( ) is a routine that has been part of my standard library for some time. When DOS is called to write a file and the number of bytes specified is zero, DOS truncates the file at that point. For reasons of safety and history, the C *write( )* routines doesn't do this; it returns zero bytes written. I can't really argue with that philosophy, but sometimes you must truncate a file. In the case of the File class, the *write( )* method continues to behave as expected and returns immediately, with no effect, if zero bytes are written. An explicit message to a File object to truncate though, uses DOS to actually chop off the file at the current file pointer location.

Set_ position( ) and *get_ position( )* allow a File client to specify where the file pointer should be. *Size( ), rename( ), erase( ),* and *copy( )* do what their names suggest. Notice that both *copy( )* and *rename( )* operate whether the file is open or not. They both preserve the current condition, restore upon completion, and, aside from the long-winded error-testing stuff, they're all simple.

Epilogue

An obvious heir of File is class *Buf_File* that would provide buffered I/O. This would be very easily accomplished by descending publicly from class File and writing new *read( )* and *write( )* methods. Nothing else is required. A class *Text_File* might in turn descend from *Buf_File.* A Directory class could descend directly from *File_Spec.* C++ 2.0 offers multiple inheritance and the possibilities boggle the mind.

I chose C++ for my OOP explorations thinking that my familiarity with C would enable me to clearly see what OOP was against — the backdrop of a known language. In retrospect I think it had the opposite effect.

In many ways object-oriented programming is like writing several mini-programs. This has the advantage of restricting the problem domain at any given point to a more easily managed size. I also found that if one thinks of objects in this way they are easier to define. For a lucid description of the concepts of OOP, I highly recommend Bertrand Meyer's book, *Object-oriented Software Construction* (Prentice Hall, 1988), which does an outstanding job of describing what OOP is and isn't and, although the book uses Eiffel (which Meyer wrote), it helped my C++ programming immensely. C++ in turn is having an impact on my C programming.

OOP's promise is to bring us a bit closer to component level design and implementation. I think OOP does have the potential to accomplish this. The cost is another layer of code between you and the machine. This means slower and larger programs, in other words, less efficient use of computer resources. On the other hand, the extra layer of abstraction will mean more efficient use of the programmer. No language can decide this swap-off for us. There will always be a need and a time for assembler, just as there will always be a need and a time for dBase IV.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal,* 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 96.)

Vote for your favorite feature/article.
Circle Reader Service **No. 5.**

A Pixel Ordering Algorithm

A shortcut for interactive development

Norton T. Allen

ersonal computers can generate and display some very interesting graphics. Unfortunately, some of the most interesting programs require hours to produce a single graphics screen. Mandelbrot sets and ray-tracing programs do extensive floating-point calculations for each pixel of the final display. The iterations can be painfully slow if the resulting image needs touching up. Often, a general idea of how the final image will look is enough to act on, but the standard linear approach to drawing the screen can mean a long wait just to find out what's going on in the lower right-hand corner.

For programs that produce graphics one pixel at a time, a better method of pixel ordering provides a shortcut during interactive development. By selecting pixels that are always evenly distributed on the screen, the general character of the final image can be seen early on. Subsequent calculations continuously improve the focus until full resolution is achieved. This makes it possible to jump in early and adjust colors, move objects, or zoom in on a particular area of interest. I have implemented this algorithm for viewing the Mandelbrot set, but it also applies as well to ray-tracing or any other pixel-by-pixel task.

Reversing Bit Order

Part of my job is writing software to receive data transmitted from high-altitude research balloons. Information from many different sensors is collected on the balloon, sorted into a standard data structure, and then broadcast as a serial bit stream. On the ground, the process is reversed. Bits are collected into bytes, and the bytes become members of structures. To help sort things out, we use a frame counter in the data stream. This is a 1- or 2-byte integer that increments every time it is transmitted. Once we successfully locked onto a data stream, but the frame counter readout showed no apparent pattern. By changing the display to binary, the problem became obvious — the normal bit order (MSB – >LSB) was reversed (LSB – >MSB in our case).

Z	bit-reversed	pixel #	Pattern
000	000	0	
001	100	4	
010	010	2	
011	110	6	
100	001	1	
101	101	5	
110	011	3	
111	111	7	

Figure 1: *Bit-reversed numbers can be used to address pixels in a row*

map = 1010_2

Z	X	Y	Pattern
0000	00	00	
0001	10	00	
0010	00	10	
0011	10	10	
0100	01	00	
0101	11	00	
0110	01	10	
0111	11	10	
1000	00	01	
1001	10	01	
1010	00	11	
1011	10	11	
1100	01	01	
1101	11	01	
1110	01	11	
1111	11	11	

Figure 2: *The results of combining both indices in a single counter by interleaving the bits of their binary representations in reverse order*

In that case, bit-reversal was the root of the problem. When I began thinking about generating pixel-oriented graphics, however, bit-reversal was the key to a solution. When counting the binary integers from 000_2 to 111_2, the most significant bit on the left is not set until the halfway point at 100_2. If the bits are reversed, the least significant bit is not set until halfway through, and all bit-reversed numbers up to that point are even. As Figure 1 shows, these bit-reversed numbers can be used to address a row of pixels. Only even pixels are addressed until $Z=100_2$. At each step, the distribution of filled and unfilled pixels is as uniform as possible.

This is fine for addressing a single row of pixels, but both rows and columns need to be addressed. Two independent counters in a nested loop could be used, but this would not give the desired result. The inside loop would completely fill one row before the next evenly-spaced row was selected.

The following algorithm combines both indices/coordinates *(X, Y)* in a single counter, *Z*, by interleaving the bits of their binary representations in reverse order. The variable *map* (shown in Figure 2) keeps track of which bits correspond to which coordinate (*X* or *Y*). Bits in *Z*, which correspond to zero bits in the *map* are part of the *X* coordinate. Bits corresponding to one are

Norton is a systems analyst/programmer with the Harvard University Atmospheric Research Project and can be reached at 202 Ridge St., Winchester, MA 01890.

part of the *Y* coordinate. The results are demonstrated in a 4 × 4 region in Figure 2.

This is close to the desired result, but the distribution at the halfway point (shown in the second pattern in Figure 2) could be more uniform. Half the rows are completely filled in, while the other half haven't been filled in at all. This problem can be seen in a 2 × 2 example. The algorithm described produces *X,Y* pairs in the order *(0,0), (1, 0), (0, 1), (1, 1)*, which can be represented:

0	1
2	3

A more desirable *X,Y* pair ordering is *(0,0), (1,1), (0,1), (1,0)*, or:

0	3
2	1

The *X* coordinates for both orders are the same, but the *Y* coordinates are different. The following inset shows a truth table comparing the original order (X_Z, Y_Z), and the new order (X_Z, Y'_Z).

	Y'_Z \ X_Z	0	1
Y_Z	0	0	1
	1	1	0

The truth table shows that $Y'_Z = X_Z \oplus Y_Z$. ⊕ designates the exclusive-OR operation. Once X_Z and Y_Z are calculated, it is easy to generate Y'_Z and report the new point (X_Z, Y'_Z). Figure 3 shows how this works in a 4 × 4 case.

I have implemented this algorithm for viewing the Mandelbrot set, but it would apply as well to ray-tracing or any other pixel-by-pixel task

There are a few rough edges dealing with arbitrary rectangular regions. First, the ranges may not be even powers of two. This is handled by rounding up to the next even power of two and then discarding any points outside of the rectangular region. Second, even after rounding, the regions may not be square. One coordinate may have more bits than the other. This is handled by storing the extra bits on the LSB end of *map*. If *X* has 2 bits and *Y* has 4 bits, $map = 101011_2$. The exclusive-OR operation imposes an additional complication. The *Y* range must be at least as large as the *X* range. This is handled by reversing the roles of the two coordinates, if the *X* range is larger.

Compilation

I have implemented a Mandelbrot set

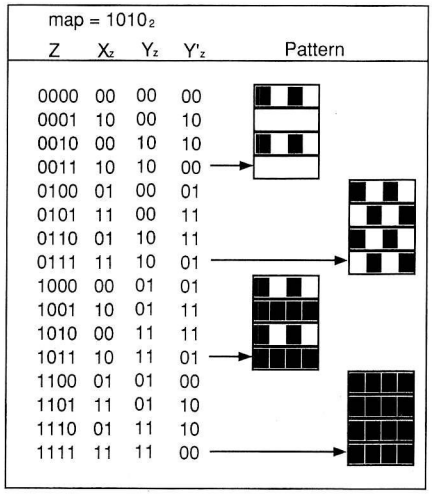


Figure 3: *Y'_Z is generated and the new point (X_Z, Y'_Z) is reported as shown in this 4× 4 matrix*

(continued from page 58)
example using grafix.lib — Kent Porter's graphics library (see *DDJ*, February 1989 through August 1989). This library is compatible with Microsoft C 5.1 and Turbo C 2.0. I compiled this project using Lattice C, Version 3.41, and some minor modifications were necessary. Lattice did not support mixed mode programming before Version 6.0. Grafix.lib contains explicit "far" designations. These designations are redundant when programming in the large model, and problematic when programming in the small model. Because my program is small, I removed the explicit declarations. I also ran into syntax difficulties using Kent's *#define byte unsigned char*, which vanished when I used *typedef unsigned char byte*.

On the assembly side, I modified the initializations to use the Lattice assembly macros to provide model-sensitive procedure definitions. I also followed Lattice's recommendation and saved ES when that register was used. Finally, Lattice function and data names are not prefaced with an underscore in assembly modules.

The Code

The file points.c (Listing One, page 116) contains two routines: *init_points( )* and *next_point( )*. The *X* and *Y* limits are passed to *init_points( )*, which initializes the *map* variable. *next_point( )* defines the coordinates of each successive pixel through pointer arguments, returning a non-zero value when the area has been completely filled.

Mandel.c (Listing Two, page 116) provides functions particular to the Mandelbrot example. The function *set_range( )* creates a virtual coordinate system. This function is similar to grafix' *setcoords( )*, with two major differences: The function *set_range( )* guarantees that the new coordinates preserve the aspect ratio of the screen. One inch on the screen should represent the same number of units in either the *X* or the *Y* direction. If one range is proportionately smaller than the other, the coordinates are extended to place that range in the center of the screen. The function *setcoords( )* only provides functions to convert from virtual coordinates to device coordinates, but in this program we need to do the reverse. *mandel_color( )* contains the Mandelbrot calculations to determine the color of a particular pixel.

The pixel ordering presented here is very closely related to dither matrices used for generating gray shading on monochrome monitors

Generate.c (Listing Three, page 116) provides the operational framework for the demonstration program and point.h (Listing Four, page 118) is the include file. *generate( )* calls *next_point( )* to step through the pixels, and calls *mandel_color( )* to determine their color. *menu( )* is called any time a key is pressed. *menu( )* lets the user move and scale a rectangular box on the screen to identify a region for closer examination. The area in the rectangular box can be focused or zoomed on. Focusing confines subsequent calculations to the bounds of the box. This gives a sharper look at a small area, in the context of the larger screen. Zooming in rescales the virtual coordinates of the screen to the coordinates of the box.

This example demonstrates the advantages of this algorithm. A small region can be zoomed in without waiting for the completion of intermediate views. By displaying the full scale image at low resolution, the zooming can be accurately selected. Focusing can be used to choose from different regions. The beauty of this algorithm is that the same amount of detail is provided in the same amount of time, re-

gardless of whether the image is zoomed or focused. If you zoom, a longer wait will produce even more detail.

XOR

To overlay a movable box on the graphics display, a significant addition was made to the library. I added the ability to draw pixels on the screen by calculating the exclusive-OR (XOR) of the current foreground color and the color already on the screen. XOR is very useful for three reasons. If you XOR a bit pattern with a pattern of ones you get the ones complement of the original bit pattern. Whether writing to a black or a white background, a change is visible. XORing a bit pattern with zeros has no effect. If a bit pattern is XORed with itself, the result is zero. XOR is associative. If bit pattern X is XORed with non-zero mask M, a result will be seen. If the result is XORed with M again, the result is $(X \oplus M) \oplus M$. This is equal to $X \oplus (M \oplus M)$, which is equal to $X \oplus 0$, which is equal to X.

XOR allows an image to be put on and removed from the screen quickly without destroying the image being written over. This is very useful for interactive applications. Not only does it allow the cursor to move around a graphics screen, but it also enables experimentation with the placement of objects before committing them to an image.

This is so useful that it is implemented in the hardware of the EGA adapter. The write function register can be set to replace the stored value, or to perform *OR, AND,* or *XOR* functions between the stored value and the value being written. In this case, I want to move a rectangular box. The function to draw the box is *draw_rect( )*. This function calls *draw_line( )*, which calls *draw_point( )*. Adding a XOR mode to *draw_point( )* allows the box to be moved. Listing Five, page 118, contains definitions to be included in grafix.h, and Listing Six, page 118, contains wmode.c, which defines the function *set_write_mode( )*. This module should be compiled and included in grafix.lib by using the *LIB grafix +wmode;* command.

Two changes need to be made in drawpt.asm. Below line 14, where the *vuport* pointer is declared, add the line: *EXTRN _write_mode : BYTE.* Also change line 78 from: *xor ah, ah* to: *mov ah, _write_mode.* Assemble *drawpt*, then replace it in the library with the command: *LIB grafix -+drawpt.* A similar change can be made to hline.asm by adding the declaration after line 14, and changing line 106.

Other Applications

To use this algorithm in a ray-tracing study, replace the functions in mandel.c with the ray-tracing code and change the calls in *generate( )* accordingly. Another application of this algorithm is for demo programs, where one graphics image fades into another. Running the fade one pixel at a time is much too slow. Divide the image into small rectangles, use this algorithm to index the rectangles, and use a *bitblt* routine to copy the rectangles.

The pixel ordering presented here is closely related to dither matrices used for generating gray shading on monochrome monitors. For 16 shades of gray, the screen is tiled with imaginary 4×4 pixel blocks. Each pixel is numbered from 0 to 15, according to its position in the order. When the user wishes to color an area with a certain gray level, only those pixels with numbers less than that level are turned on.

Tradeoffs

This algorithm does much bit twiddling to calculate each set of coordinates. This adds CPU processing time. On the 4.77-MHz XT, this algorithm took 15 minutes longer to fill the screen than a simple nested loop. If the exact calculations for drawing an image are already known, it is preferable not to use this algorithm. When a lot of CPU time is being used, however, it's nice to know if the right picture is being generated. Seeing only the left quarter of the screen won't help, but one quarter resolution across the entire display is enough to see considerable detail.

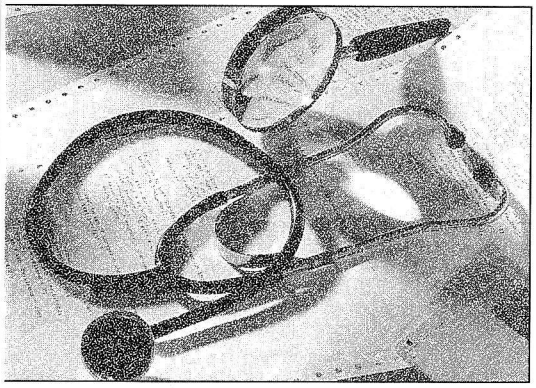
Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 116.)

Vote for your favorite feature/article.
Circle Reader Service **No. 6.**



Examining Instant-C

Exploring protected mode with an interactive environment

Andrew Schulman

Instant-C is an interactive C compiler and integrated development environment from Rational Systems, based on Rational's DOS/16M, a protected-mode DOS extender for Intel 80286- and 80386-based PC compatibles. DOS/16M is also used in such products as Lotus 1-2-3, Release 3, AutoCAD, Release 10.0, and the DOS version of the Glockenspiel C++ compiler.

Instant-C (IC) provides interactive execution, linking, editing, and debugging of C code. In addition to loading .C files, C expressions can be typed in at IC's # prompt for immediate evaluation.

Figure 1 shows a sample session with IC. First, a buffer is allocated with *malloc( )*. This buffer happens to reside in extended memory, as shown by a call to the DOS/16M function *D16AbsAddress( )*. As with any product based on a DOS extender such as DOS/16M, however, the distinction between extended and conventional memory is largely unimportant: In protected mode, it's all just memory.

Next, the low-level Microsoft C *dos_open( )* function is used to open a file, and *dos_read( )* is used to read the file into our buffer. We could just as easily use the C standard library functions *fopen( )* and *fread( )*, but using these DOS-specific routines in conjunction with the extended-memory buffer shows how a DOS extender transparently manages the interface between MS-DOS and protected mode.

Note how C statements, declarations, and preprocessor statements can be freely mixed at the # prompt, somewhat like mixing statements and declarations in C++. In this "immediate mode," leaving the semicolon off a statement tells IC to print its value. This is one way that interactive C differs from "normal" C. In Figure 1, I displayed the value of the variable *bytes* simply by typing its name.

```
# char *p;
# #include <malloc.h>
MALLOC.H included
# p = malloc(10240)
      address 03B8:01A6:" "
# #include "dos16.h"
DOS16.H included
# D16AbsAddress(p)
      2940246 (0x2CDD56)
# #include <dos.h>
DOS.H included
# int handle;
# _dos_open("\\msc\\inc\\dos.h", 0, &handle)
      0
# handle
      7
# unsigned bytes;
# _dos_read(handle p, 10240, &bytes)
      0
# bytes
      5917 (0x171D)
# _dos_close(handle)
      0
# printf("%s\n", p)       // display file
```

Figure 1: *A sample session with Instant-C*

IC provides a command language in the form of preprocessor statements. To compile a file FOO.C, for example, you could type *#load foo.c* at the IC prompt. The command language can also be used under program control:

```
if (x > 1)
    _interpret("#load foo.c");
```

Working with a "quick" environment raises the issue of compatibility with production compilers. IC has a standard library, but for using the Microsoft C library instead, for example, IC comes with scripts to load MSC 5.1's real-mode large-model LLIBCE.LIB, load IC-supplied .LIB modules to replace the few Microsoft functions that won't work in protected mode, *#include* the Microsoft header files into IC, and then write out a new, large model, MSC-compatible IC. The new executable not only runs your C programs in protected mode under MS-DOS: It executes the Microsoft C library in protected mode as well. It seems like quite an accomplishment to load real-mode object code and execute it in protected mode, but this is standard procedure for DOS extenders.

IC gives C the interactive style of languages such as Forth and Lisp. Contrary to the stereotype of an interpreter, IC uses native object code. In fact, IC

Andrew Schulman is a software engineer who works on networking software for CD-ROM. He is a contributing editor of DDJ, *and a coauthor of the book* Extending DOS *(edited by Ray Duncan, Addison-Wesley, May 1990), from which this article is adapted. Andrew can be reached at 32 Andrew St., Cambridge, MA 02139.*

(continued from page 62)
can dynamically load and link .OBJ and .LIB files, and can write out stand-alone .EXE files. Such stand-alone executables include a built-in protected-mode DOS extender.

The two major benefits of protected mode — memory protection and a large address space — mesh with the needs of a C development environment. The large address space (up to 16 Mbytes of memory) means that even very large C programs can be developed interactively. Hardware-based memory protection also helps insulate IC from bugs in user code and assists IC in finding bugs. An interpreter running in protected mode can off-load some of its type-checking onto the CPU.

IC is not only a product built using a DOS extender, it is an example of why "EXTDOS" is necessary in the first place. IC has been in existence since 1984. As more and more features were added to the product, it began to strain against the artificial 640K "Berlin Wall" of real-mode MS-DOS. Rational Systems developed DOS/16M for IC to cope with its expanding features and resulting expanding memory consumption. Thus, DOS/16M is based on IC, as much as IC is based on DOS/16M. For a short time, Rational Systems marketed a separate protected-mode IC/16M alongside real-mode IC. In October 1989, with IC Version 4.0, Rational discontinued the real-mode version.

Using Protection

A protected-mode interpreter must allow the user to violate the CPU's protection model without causing the interpreter itself to be shut down. I have discussed this issue at length in my two-part article, "Stalking GP Faults" (*DDJ*, January 1990 and February 1990). In IC we can freely type protection violations at the # prompt, as shown in Figure 2, because IC installs its own general-protection violation (GP fault) handler.

In addition to helping find bugs during development, the hardware-based memory protection of the Intel processors also can be put to work in the deliverable version of a product.

For example, functions often perform their own range checking. Each time the function is called, its parameters are checked against the size of the target object. But because the hardware does range and type checking in protected mode anyway, and because we pay a performance penalty for this checking, we should get the hardware to do our checking as well.

This requires devoting a separate selector to each object for which you want hardware-assisted checking. To see why, let's overstep the bounds of an array and see whether the Intel processor detects an off-by-one fencepost error. In Figure 3, the pointer *p* points to a block of 2013 bytes, numbered 0 through 2012, so *p[2013]* clearly oversteps its bounds. If protected mode is all it's cracked up to be, the CPU should have complained, right? Why didn't it?

The reason is that *malloc( )* and other high-level-language memory allocators suballocate out of pools of storage. They do not ask the operating system for memory each time you ask them for memory, nor would you want them to. From one segment, *malloc( )* may

```
# char *s;     // oops: forgot to initialize
# atoi(s)
  ## 492: Invalid address 0AA8:4582 at __CATOX+000E
# #reset
```

Figure 2: *Detecting a protection violation*

(continued from page 64)
allocate several different objects. While we think of the object *p* as containing 2013 bytes, the processor sees a considerably larger object: The block of memory *malloc( )* received the last time it asked DOS for memory. What size is the object the CPU sees?

```
# D16SegLimit(p)
   24575 (0x5FFF)
```

If this explanation is correct, trying to poke *p[0x5FFF]* ought to cause a GP fault:

```
# p[0x5fff] = 'x'
   ## 492: Invalid address 0BF8:002A
```

Now, we still need a way to make the CPU see things our way. Because 80286 memory protection is based on segmentation, we must devote a separate selector to each object for which we want hardware-assisted checking. Notice I said "selector" and not "segment." We can continue to use *malloc( )* to allocate objects, but when we want the CPU to know how big we think the object is, we provide an "alias" in the form of another selector that points at the same physical memory but whose limit is smaller. The segment limit indicates the highest legal offset within a block of memory and is checked by the CPU for each memory access.

To create an alias *q* for the pointer *p*, where *q*'s limit is equivalent to the array bounds, we can use two other DOS/16M functions, as shown in Figure 4. This takes a physical address returned by *D16AbsAddress( )*, together with the limit we're imposing for access to this memory, and passes them to *D16SegAbsolute( )*, which constructs a protected-mode selector for the same absolute physical address but with a different limit.

In Figure 5, we verify that this worked. Attempting even to read from this invalid array index now causes a GP fault.

So we can dispense with explicit

```
# char *p;
# p = malloc(2013)
     address 03B8:01A6: " "
# p[2013] = 'x'
     'x' (0x78)
```

Figure 3: *This off-by-one error doesn't violate protection*

```
# char *q;
# q = D16SegAbsolute(D16AbsAddress(p),
                                  2013);
   0C08:0000
```

Figure 4: *Creating an alias with a shorter limit*

bounds checking: The CPU will check for us. To control the error message displayed when a GP fault occurs, we could write our own INT 0D handler and install it using the DOS set-vector function (*INT 21 AH=25*). Thus, instead of littering error checking throughout our code, protected mode allows us to centralize it inside an interrupt handler. Errors can be handled after the fact, rather than up front. In a way, this resembles ON ERROR, one of the more powerful concepts in Basic (which got it from PL/I).

This meshes with the advice given by advocates of object-oriented programming: "If you are expecting a sermon telling you to improve your software's reliability by adding a lot of consistency checks, you are in for a few surprises. I suggest that one should usually check less. . . . 'Defensive programming' is a dangerous practice that defeats the very purpose it tries to achieve" (Bertrand Meyer, "Writing Correct Software," *Dr. Dobb's Journal*, December 1989).

By using protection, you may be able to make an application run faster in protected mode than under real mode, because a lot of error-checking and "paranoia" code can now be made unnecessary.

When finished with the pointer *p*, it is important not only to *free(p)* but to release the alias in *q*. Don't use *free( )* to release this selector, though: The C *malloc( )* manager doesn't know anything about *q*, which is just an alias, a slot in a protected-mode descriptor table. We need to free this slot because the number of selectors available in protected mode is quite limited:

```
# free(p)
# D16SegCancel(q)
```

In moving from real to protected mode, programmers may regret that segment arithmetic is so restricted. But the ability to create aliases, different views of the same block of physical memory, means that protected-mode selector manipulation is actually far more versatile than real-mode segment arithmetic.

The Intel 286 Protected-Mode Instructions

Transparency is a major goal of DOS extenders. But sometimes it is useful not to be so transparent. For example, DOS extender diagnostic programs and DOS extender utilities will generally be nonportable, hyper-aware that they are running in protected mode.

The 80286, and also the 286-compatible 386 and 486 chips, have a number of instructions that Intel provides primarily for use by protected-mode operating systems but which are also useful for utilities and diagnostic programs. Some of these are listed in Figure 6.

For example, in the last section we called *D16SegLimit( )* to find the size of the segments pointed to by *p* and *q*.

```
# q[2013]
  ## 492: Invalid address 0BF8:002A in command line
# D16AbsAddress(p) = = D16AbsAddress(q)
    1
# D16SegLimit(p) = = D16SegLimit(q)
    0
# D16SegLimit(p)
    24575 (0×5FFF)
# D16SegLimit(q)
    2012 (0×7DC)
```

Figure 5: *Now the off-by-one error does violate protection*

In operation (though not in implementation), *D16SegLimit( )* corresponds to the LSL instruction, which takes a selector in the source operand and, if the selector is valid, returns its limit (size -1) in the destination operand. For example:

```
lsl ax, [bp+6]
```

The two major benefits of protected mode—memory protection and a large address space—mesh with the needs of a C development environment

Similarly, the LAR instruction will load the destination operand with the "access rights" of the selector in the source operand if it contains a valid selector:

```
lar ax, [bp+6]
```

The instructions LSL, LAR, VERR, and VERW are special because, even if the selector in the source operand is not valid, the instructions don't GP fault; instead, the zero flag is cleared. Therefore, if these instructions were available in a high-level language, we could construct protected-mode memory browsers and other utilities simply by looping over all possible selectors. This is an odd form of segment arithmetic:

```
for (i=0; i<0xFFFF; i++)
    if lar(i) is valid
        print_selector(i)
```

It is easy to make the Intel protected-mode instructions available to C and other high-level languages, and they can be used interactively in IC. PROTMODE.ASM (Listing One, page 120) is a small library of functions, including *lsl( )* and *lar( )*, that can be assembled into PROTMODE.OBJ by using either the Microsoft Assembler (Version 5.0 and later) or Turbo Assembler.

PROTMODE.ASM uses the DOSSEG directive, which simplifies writing assembly-language subroutines and uses the ENTER and LEAVE instructions provided on the 80286 and higher for working with high-level-language stack frames. These execute a little slower than the standard BP-SP prolog/epilog but create compact source code.

PROTMODE.ASM provides nothing more than a functional interface to the Intel protected-mode instructions. While completely nonportable with real mode, this module is highly portable among 16-bit protected-mode systems (it would require some modification for use with a 32-bit DOS extender). Once assembled into PROTMODE.OBJ, it can be linked into any 16-bit protected-mode program, including an OS/2 program. It can be loaded into IC:

```
#loadobj "protmode.obj"
```

You need to supply stub definitions for the individual routines in an object module loaded into IC. These look almost like declarations or function prototypes, except that they are followed by the construct *{extern;}*. PROTMODE.H (Listing Two, page 120) is a C *#include* file that contains function prototypes for use with IC (*#ifdef InstantC*) or with any other 16-bit protected-mode environment.

Now it's time to test the functions. Let's allocate a 10K segment, and see what limit *lsl( )* returns. Figure 7 uses the *FP_SEG( )* macro from Microsoft's dos.h to extract the selector from the pointer *p*, and passes this to *lsl( )*; *lsl( )* returns 10,239, which is clearly the last legal offset within a 10K segment, so *lsl( )* seems to work. (Actually, there is an extremely obscure bug in *lsl( )*. You should be able to spot it by looking back over Listing One.)

The *verw( )* function, like the VERW instruction, returns TRUE if a selector can be written to, or FALSE if the selector is read-only:

```
# verw(FP_SEG(p))
    1
```

LSL	(load segment limit) —size of a segment
LAR	(load access rights) —access rights of segment
VERR	(verify read) —can segment be peeked?
VERW	(verify write) —can segment be poked?
SGDT	(store GDT) —base address and size of GDT
SIDT	(store IDT) —base addr and size of IDT
SLDT	(store LDT) —selector to LDT

Figure 6: *Some Intel protected-mode instructions*

```
# char *p;
# p = D16MemAlloc(10240)
    address 0C08:0000
# lsl(FP_SEG(p))
    10239 (0x27FF)
```

Figure 7: *Verify that* lsl( ) *works*

We can use a DOS/16M function to mark this segment as read-only and then see if *verw( )* has picked up on the change in the selector attributes:

```
# D16SegProtect(p, 1)
   0
# verw(FP_SEG(p))
   0
```

The read-only attribute, like other aspects of the protected-mode "access rights," applies to a selector, not to the underlying block of memory. One selector can be read-only and another read/write, while both correspond to the same physical memory.

Having tested the PROTMODE.OBJ routines we can, as promised, write a simple loop to display all valid selectors within our program. In IC, of course,

Instant-C gives C the interactive style of languages such as Forth and Lisp

we can just type this in at the # prompt, as shown in Figure 8.

This will display all valid selectors within a protected-mode program (not just a DOS/16M program). But to be genuinely useful we need to print out some additional information about the selectors. In addition to using several of the functions in PROTMODE.ASM, the code in BROWSE.C (Listing Three, page 120) also performs some manipulations on the selector number itself: The bottom two bits are extracted with the expression *i & 3*, and the third bit is extracted with the expression *i & 4*.

What?! A protected-mode selector, unlike a real-mode segment number, has no necessary relation to the segment's physical location in memory. A protected-mode selector closely resembles a file handle. It is almost a "magic cookie," but not exactly, in that the number itself actually has semantic meaning: A selector is a record comprised of three fields. The bottom two bits contain a protection level, zero (most privileged) through three (least privileged). The third bit from the right contains a "table indicator" — zero means the selector belongs to the Global Descriptor Table (GDT), and one means it belongs to the Local Descriptor Table

(continued from page 70)
(LDT) — and the remaining 13 bits form an index into this table. Thus, when applied to a protected-mode selector *i*, *i & 3* extracts the selector's protection level, and *i & 4* tells whether the selector is located in the GDT or LDT.

Running under IC, a small part of the output from BROWSE.C is shown in Figure 9. The list runs on and on for quite a while. What is the value of this?

In contrast to real mode where every address you can form points somewhere, protected-mode memory is a sparse matrix. At any given time, most segment:offset combinations are not valid addresses: Dereferencing them causes a protection violation. Producing a list like this gives us an idea of the memory organization of a DOS extender program.

From this list, we can see that while protected-mode memory is a sparse matrix, it's not so sparse under DOS/16M as under OS/2. We can also see that all the entries are marked PL=00, indicating that everything is running at Ring 0. To double-check that this is so, the loop in Figure 10 represents the query, "Are any segments not at Ring 0?". Under IC, this produces no output: Everything is running at the most privileged protection level. But in OS/2, most of an application program's selectors would be displayed by this loop. This is one of the differences between DOS/16M and a full-blown protected-mode operating system such as OS/2. Because DOS/16M is just a shell to support one program at a time in protected mode, Rational Systems chose not to establish different protection levels.

Along the same lines, the selectors you'll use in IC or in DOS/16M actually refer not to your program's LDT, but to the GDT. Because there is only one program running, the distinction between GDT and LDT, while crucial in a multitasking operating system such as OS/2, is fairly artificial in the "one program at a time" world of DOS/16M.

On the other hand, another DOS extender, Eclipse Computer Solution's OS/286, while sharing many of the same goals as DOS/16M, makes a sharper distinction between the kernel (the OS/286 DOS extender itself) and the program supported by the DOS extender. OS/286 programs run at Ring 3, while OS/286 itself runs at Ring 0. This just shows that there are few fixed rules about how a DOS extender must be organized. Protected mode allows for a wide variety of styles in operating environments.

IC requires a large GDT partially to support many "transparent" selectors. For example, selector 0x40 has a physical base address of 0x400, corresponding to the BIOS data area. Using the same code from PROTMODE.ASM, it is trivial to form the query, "Which selectors are transparent?" (Figure 11.)

```
unsigned i;
for (i=0; i<0xFFFF; i++)          // for all possible selectors
   if (lar(i))                    // if a valid selector
      printf("%04X \ n", i);      // print selector
```

Figure 8: *Display all valid selectors*

```
0038    000000    LAR=93  LSL=FFFF  PL=00   VERR  VERW  GDT
003C    000000    LAR=93  LSL=FFFF  PL=00   VERR  VERW  LDT
0040    000400    LAR=93  LSL=0FFF  PL=00   VERR  VERW  GDT  TRANS
0044    000400    LAR=93  LSL=0FFF  PL=00   VERR  VERW  LDT
0048    034FE0    LAR=93  LSL=FFFF  PL=00   VERR  VERW  GDT
004C    034FE0    LAR=93  LSL=FFFF  PL=00   VERR  VERW  LDT
0050    110010    LAR=93  LSL=200F  PL=00   VERR  VERW  GDT
0054    110010    LAR=93  LSL=200F  PL=00   VERR  VERW  LDT
0058    032050    LAR=81  LSL=0067  PL=00               GDT
005C    032050    LAR=81  LSL=0067  PL=00               LDT
0060    FA0000    LAR=93  LSL=FFFF  PL=00   VERR  VERW  GDT
0064    FA0000    LAR=93  LSL=FFFF  PL=00   VERR  VERW  LDT
0068    100010    LAR=82  LSL=FFF8  PL=00               GDT
```

Figure 9: *Output from BROWSE.C*

```
for (i=0; i<0xFFFF; i++)
   if (lar(i) && (i & 3))
      printf("%04X PL=%02X\n", i, i & 3);
```

Figure 10: *Are any selectors not at Ring 0?*

```
for (i=0; i<0xFFFF; i++)
   if (lar(i) && (i == D16Abs Address
                     (MK_FP(i,0)) >> 4))
      printf("%04X ", i);
```

Figure 11: *Are any selectors transparent?*

```
typedef struct {
      unsigned        limit, lo;
      unsigned char hi, reserved;
      } GDTR;
```

Figure 12: *The GDTR represented in C*

```
# GDTR g;
# sgdt(&g)
# g
    struct  at 2F1C {
        limit = 65528 (0xFFF8);
        lo = 16 (0x10);
        hi = '\020' (0x10);
        reserved = '\0';}
```

Figure 13: *Finding the location and size of the GDT*

Examining the Protected-Mode Descriptor Tables

We have already used an indirect method to examine the DOS/16M memory map: Loop over all possible selectors and see if they're legal. We can also directly examine the GDT, IDT, and LDT.

PROTMODE.ASM contains a functional interface to the SGDT instruction. SGDT expects a pointer to 6 bytes of storage (a FWORD PTR), into which it copies the contents of the CPU's GDT register (GDTR). The GDTR holds the 24-bit physical base address and 16-bit limit of the GDT, corresponding to the C structure in Figure 12. (Note that this, like most structures in this article, requires byte alignment; in *IC, _struct _alignment = 1*; in batch compilers such as Microsoft C, use *#pragma pack (1).*) This structure, along with *sgdt( )*, is used in Figure 13 to get the physical base address (0x100010) and limit (0xFFF8) of the GDT.

Now we need to map this into our address space. A protected-mode descriptor table is an array of 8-byte segment descriptors. Each descriptor contains the 24-bit physical base address and 16-bit limit for the segment, as well as an access-rights byte. There is also a 2-byte field used in 32-bit protected mode on the 386. All this can be expressed in C, as shown in Figure 14. After typing or loading this structure definition into IC, we can create a pointer to the GDT (Figure 15).

Now that we have a pointer to the GDT, let's make it read-only to make sure we don't mess anything up (though, if you were working in a protected-mode environment that didn't have convenient functions for changing selector attributes, you might actually want to write to the GDT!):

```
# D16SegProtect(gdt, 1);
```

If bit 3 of a selector indicates that it

(continued from page 72)
belongs to the GDT, then the top 13 bits of the selector can be used as an index into the GDT. Take the example of the GDT pointer itself. In Figure 16, we locate our new descriptor for the GDT within the GDT. (Got it?)

Figure 16 confirms what we already know about the GDT: Its physical base address is 0x100010 and its limit is 0xFFF7. We could now dispense with *D16SegAbsolute( )* and *D16SegLimit( )*, and write portable protected-mode code.

To get a pointer to the GDT generally requires that you use some special facility within your protected-mode environment. We used *D16SegAbsolute( )* here, which obviously won't work outside DOS/16M. However, once you do have a pointer to the GDT, you can write completely portable protected-mode code. For example, I will snarf a lot of this code for a forthcoming *DDJ* article that features a GDT browser for OS/2.

What about the "access rights" value that the CPU in protected mode uses to ensure proper use of selector 0x0C08? We can use the C bitfield in Figure 17 to display the individual fields that make up the access-rights value 0x91. The C bit field structure is wonderfully nonportable, so if using the structure in Figure 17, you should check your compiler's ordering of bit fields and make sure the stucture is *byte* aligned.

While DOS/16M (and, consequently, IC) doesn't make much use of the LDT, this table is crucial in other protected-mode environments. Getting the LDT selector is simple:

```
DESCRIPTOR far *ldt;
ldt = MK_FP(sldt(), 0);
```

If this pointer is not valid within your address space (*!verr(sldt( ))*), you can instead look up your LDT's descriptor within the GDT:

```
gdt[sldt( ) >> 3]
```

and then map the absolute address you find there into your address space.

Now that we have these structures, we can write a function, *sel( )*, to display selector attributes. Note that in Listing Four, page 120, (SEL.C) contains no references to DOS/16M. *sel( )* can be used to examine the selector for any pointer, such as *sel( )*'s own function pointer. The attributes display indicates that this is readable code running at protection level zero:

```
# sel(sel)
SEL=0A68 ADDR=472BB0
        LIMIT=43FF ACCESS=0ar-c-p
```

All these data structures are described in the Intel literature on 286 and 386 protected mode. Seeing them come to life in IC, though, is a great aid to understanding protected mode.

One of the tricks of protected-mode programming is to acquire an in-depth knowledge of these data structures and then, when programming, to forget about them. The operating environment takes care of maintaining the GDT, the descriptors within the GDT, and the access-rights bytes within the descriptors. The CPU will take care of using these data structures to maintain the integrity of the system. You're better off not thinking too closely about them, but it does seem to help to have been familiar with them at some point or other. Having an interactive environment like IC is a great aid to gaining this familiarity.

```
typedef struct {
    unsigned          limit;      // size minus 1
    unsigned          addr_lo;    // physical base addr - paragraph.byte
    unsigned char     addr_hi;    // physical base addr - megabyte
    unsigned char     access;     // see ACCESS_RIGHTS below
    unsigned          reserved;   // for 386 (32-bit)
    } DESCRIPTOR;
```

Figure 14: *The protected-mode descriptor represented in C*

```
# DESCRIPTOR *gdt;      // GDT is array of DESCRIPTOR
# gdt = D16SegAbsolute((long) MK_FP(g.hi, g.lo), g.limit + 1)
      address 0C08:0000
```

Figure 15: *Mapping the GDT into our address space*

```
# gdt[FP_SEG(gdt) >> 3]
    struct at 0C08:0C08 {
      limit = 65527 (0×FFF7);
      addr_lo = 16 (0×10);
      addr_hi = '\020' (0×10);
      access = 'Q' (0×91);
      reserved = 0;}
```

Figure 16: *Finding the GDT selector within the GDT*

```
typedef struct access {
unsigned accessed   : 1;    // has segment been accessed?
unsigned read_write : 1;    // if data 1=write; if code 1=read
unsigned conf_exp   : 1;    // expansion direction
unsigned code_data  : 1;    // 0 = data, 1 = code
unsigned xsystem    : 1;    // 0 = system descriptor
unsigned dpl        : 2;    // protection level: 0..3
unsigned present    : 1;    // is segment in memory?
} ACCESS_RIGHTS;

# *((ACCESS_RIGHTS *) &gdt[FP_SEG(gdt) >>3].access)
        struct access at 0C08:0C0D {
    accessed : 1 = 1;       // it's been used
    read_write : 1 = 0;     // it's read-only
    conf_exp : 1 = 0;       // it's not a stack
    code_data : 1 = 0;      // it's data
    xsystem : 1 = 1;        // it's not a system descriptor
    dpl : 2 = 0;            // protection level 0
    present : 1 = 1;}       // it's present in memory
```

Figure 17: *The protected-mode access-rights byte in C*

Product Information

Instant-C
Rational Systems Inc.
220 N. Main St.
Natick, MA 01760
508-653-6006
Requires DOS 2.0 or above
an 80286 or 80386 CPU
and at least 1 Mbyte of memory.
Works with medium and
large memory models.
Price: $795

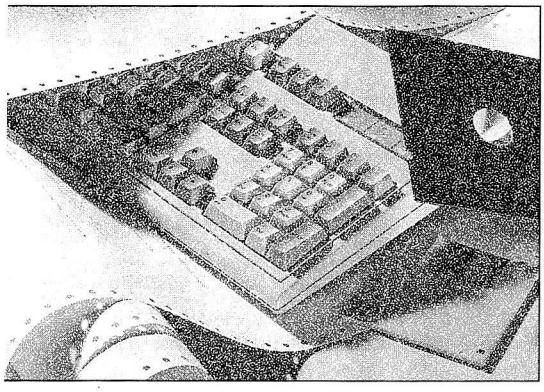
Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 120.)

Vote for your favorite feature/article.
Circle Reader Service **No. 7.**



Accessing Hardware from 80386 Protected Mode Part II

A 4-gigabyte memory model and features that control 80386 paging make FAR pointers obsolete

Stephen Fried

Last month, in Part I, we saw that with a few tricks such as tiling a huge model, and the use of FAR pointers we could address up to 64 terabytes. However, I hope to convince you by the end of this article that the only use for FAR pointers in 80386 code is in operating system kernels. As a starting point, let's take a look at ports and interrupts.

The 80386 supports ports and interrupts in the same manner as the 8086, with the exception that interrupt vectors are only stored in the first 1024 bytes in real mode, while port reads and writes can result in exceptions if the user's code does not have the required protection level. From a practical standpoint, what this means in 32-bit protected mode is that the processor looks into the interrupt descriptor table (IDT) instead of the first kilobyte anytime an interrupt happens.

These two features make it possible for an operating system or environment to control what happens when a protected-mode program hits either an interrupt or attempts to write a port. This type of facility makes it possible to run DOS applications in "compatibility boxes" that simulate MS-DOS. In the case of DOS extenders, what happens during I/O operations is that port reads and writes are passed through, while interrupts get translated into MS-DOS compatible operations. These MS-DOS compatible operations are performed by DOS itself, which is activated by returning to real mode and examining a buffer area in the first 640K that has been left behind by the protected-mode portion of the DOS extender.

Stephen is the vice president of MicroWay's R&D. He is well known in the field for his PC numeric and HF chemical laser contributions. You can reach him at MicroWay Inc., P.O. Box 79, Kingston, MA 02364.

Interrupts

To place interrupts in context, let's examine how to write a short sequence that writes to the system diskette using the ROM BIOS INT 13H entry point. Digging out an eight-year-old copy of the IBM technical user's manual, we examine the parameters that have to be passed to the routine in the registers *al*, *ah*, *cl*, *ch*, *dl*, *dh*, and *es:bx*. Figure 1 shows the equivalent 8086 and 80386 versions of a section of a program that writes eight sectors at a time to a diskette.

Of course, if we were writing this code in C, we would have used the NDP C version of *int86* or *int386*, or the register-aliased variables introduced later, in conjunction with an *asm* statement. We chose the diskette routine as an example because it requires that a pointer to a buffer is passed to the ROM BIOS. Examining the code, we discover that both pieces of code are identical, except for the buffer pointers. In the protected-mode example, we use the pointer *es:ebx* instead of *es:bx*. To understand how the interface becomes 32-bit knowledgeable, it is necessary to examine what happens when an *int 13H* executes in protected mode.

The DOS extender has set up the IDT, so that when an *int 13H* occurs, the DOS extender gets interrupted and takes over. The extender first looks at the interrupt number, and if it employs a pointer of the type *es:bx*, it enters a special mode in which it moves the data pointed to by *es:ebx* into a buffer area it has reserved for itself in real

Real mode		32-bit protected mode		
mov	ax,ds	mov	ax,ds	; set up es:bx
mov	es,ax	mov	es,ax	; es = ds
mov	bx,buffer	mov	ebx,buffer	; point at buffer
mov	al,8	mov	al,8	; # of sectors
mov	ah,3	mov	ah,3	; write to diskette
mov	cl,8	mov	cl,8	; sector # 1
mov	ch,39	mov	ch,39	; track number 39
mov	dl,2	mov	dl,2	; drive # 2
mov	dh,1	mov	dh,1	; head #1
int	13H	int	13H	; call the ROM BIOS

Figure 1: *8086 and 80386 versions of a program fragment that write eight sectors at a time to a diskette*

(continued from page 78)
memory. This makes it possible to locate our protected-mode track buffer anywhere in the 32-bit segment being addressed by *ebx*, and to use buffers larger than 64K. The extender then copies the first 64K from the protected buffer into the real buffer, sets *es:bx* so that it points at the real buffer, restores all the other registers so that they were the same as when the interrupt was invoked in protected mode, enters real mode and, finally, issues an *int 13H* to invoke the ROM BIOS diskette routine.

Another benefit of this technique is that it works up to 20 percent faster in an 80486 system than techniques that use intrasegment methods

This technique works fine for buffers that are up to 64K in size. For buffers larger than 64K, the extender is forced to make multiple transfers. As a result, there is little benefit in using buffers larger than 64K when calling either a ROM BIOS or MS-DOS entry point from protected mode. In the process of benchmarking the NDP C and Fortran I/O run-time systems, we discovered that MS-DOS is the primary I/O bottleneck, and that a buffer size of 8-Kbytes gave excellent performance.

FAR Pointers

Two years ago, when we introduced NDP Fortran, the block move technique just described was the main technique used for accessing real mode memory from protected mode. We made this facility available to users with functions whose arguments included the selector of the destination segment. This worked fine, just as long as the memory being accessed was recognized and had a selector set up for it by Phar Lap. However, as our users became more sophisticated, this technique started to break down. More and more users wanted to access new memory-mapped gadgets no one had ever heard of before.

At about the same time, a number of people started talking about adopting FAR pointers to the 80386, but when we examined the problem we discovered that adding more segments was not a solution because it did not address the problem of how to create new segments to map in new devices.

About this time, Phar Lap added some calls that made it possible to control system paging. With these calls in hand, we developed a routine that enabled us to extend the size of a user's data segment while simultaneously mapping this new extension to any arbitrary, physical address.

To gain some perspective, imagine you have just bought a digitizer which is memory mapped and has a resolution of 1000 × 1000 pixels (that is, it takes up a megabyte of physical address space in your 386 AT's I/O channel). The ideal way to obtain access to this device is to pass its size and physical location to the operating system and to get back a 32-bit pointer to the device in the address space of the program. This is exactly what *mapdev( )* does. The pointer that *mapdev( )* returns is an ordinary 32-bit pointer to a piece of your program's data segment that was just magically created. It doesn't require the use of intrasegment transfers or FAR pointers, and you don't have to worry about using a block move to access the device or memory that was mapped in.

Another benefit of this technique is that it works up to 20 percent faster in an 80486 system than techniques that use intrasegment methods. The reason is that the 80486 has a preferred segment register, *ds*, for accessing data, and all NDP data accesses (by default)

```
char *mapdev();
main()
{
          int jcount = 0;
          char *scr_ptr;
          /* map the screen into the data segment */
          scr_ptr = mapdev(0xb8000,4096);
          while (jcount < 4096)
          {
                    *(scr_ptr + (jcount++)) = 0x20; /* write space*/
                    *(scr_ptr + (jcount++)) = 0x7c; /* and attribute*/
          }
}
```

Example 1: *Using* mapdev( ) *to map the screen into the protected data segment*

(continued from page 80)
are made by using *ds*.

The choice of segment registers becomes a crucial issue when it comes to Weitek. The default technique used by NDP compilers for accessing Weitek is to pass the selector 3C into the segment register *fs* and to access Weitek through *fs*. Our research to date indicates that mapping Weitek, and everything else into *ds*, pays off with big dividends. Altogether, it is possible to make a 15 – 50 percent improvement in 486/Weitek speed by correctly aligning and mapping 486 code.

The 80486 and *mapdev( )* put the final nails in the FAR pointer coffin. The next question is, "How do you port a 16-bit C application that uses FAR pointers to 386 protected mode?" FAR pointers are used in two different ways by 16-bit C programs: To identify 20-bit pointers when the program is using 16-bit pointers by default, and to access physical locations in the 20-bit 8086 address space. The first use, increasing pointer size, does not have a corresponding feature in the small memory model employed by any of the operating systems currently in use. As a result, this type of FAR pointer gets translated by the statement *define FAR*

This statement converts occurrences of FAR into a null string. When FAR pointers are used to access physical devices in the 20-bit address space of the 8086, the easiest solution is to use *mapdev( )* to map the device into the protected data segment and use the resulting pointer instead. For instance, consider the program in Example 1 to clear the screen.

For chores such as addressing the screen, note that *mapdev( )* is a much better solution than the block move we introduced earlier. It makes it possible to read or write any region of physical memory by using 32-bit pointers created by the operating system, without invoking special functions or creating buffers that have to be moved in bulk.

At this point, you should have a good feel for what memory looks like. Figure 2 shows a map of a system running in protected mode under a DOS extender without virtual memory running (the virtual map is different).

The lowest megabyte looks just like any other real map with the exception that an area of the map between the DOS extender and its I/O buffer have been made available for protected-mode code and data if paging is enabled (the default). To simplify the map, we have drawn it with paging disabled, which places the start of the protected code and data segment at the bottom of the first megabyte. Note that the code and data of the protected-mode program itself looks very similar to an ordinary small model program with the exception that this segment can grow on the top. Just above the code and data is an area identified for expansion of the address space using the *mapdev* function, which also causes selectors 0CH and 14H to grow. Finally, at the top of the map we find the Weitek segment.

Register-Aliased Variables

At this point, we have just about introduced all the interface tricks except those that let programs running in real mode access protected-mode programs, and vice versa. These types of accesses involve writing TSRs and passing data back and forth between real and protected mode, using real mode buffers. Phar Lap does a good job of explaining how to write these types of interfaces. Because of popular demand, we have written interfaces between the NDP runtime environment and Media Cybernetics' Halo and Novell's BTrieve. However, we have still not introduced the piece de resistance of interface tricks — register-aliased variables.

Functions such as *int86*, *int386*, *outp*, *outpw*, *inp*, and *inpw*, are fine for prototyping I/O routines that interface devices directly but, in the end, any developer that is worth their salt will choose to rewrite these routines in assembly language. The reason is clear: Functions such as *int86* involve the use of bulky structures and have to pass through as many as 40 lines of code every time they are called.

In searching for a technique to speed up this operation, we chose to make two simple extensions to the language:

1. Make it possible to specify which register a register variable is stored in.
2. Notify the *asm* statement about which variables were read and written on each "call" to *asm*.

These two tricks make it possible to declare 8-, 16- and 32-bit C variables that are stored in particular 80386 registers. To invoke an interrupt by using register-aliased variables, use C assignments to set up the registers and then kick off the function with an *asm* statement that contains the interrupt being used.

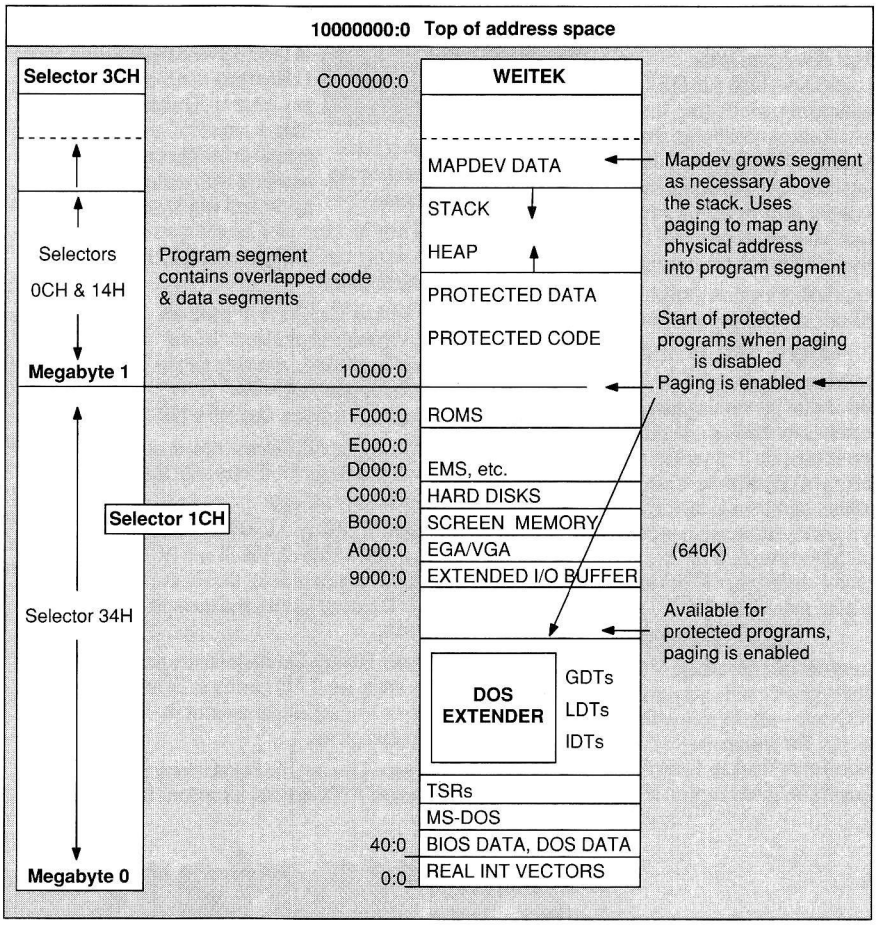


Figure 2: *Memory map of a system running in protected mode under a DOS extender without virtual memory running*

As we will see, what results is pure poetry and, frequently, much better code than can be produced by an expert assembly language programmer. A person writing an assembly routine does not know the state of all the registers (that is, which are free when a procedure is called) when the routine they are writing is called. As a result, they must save and restore all registers used (something the compiler does not have to do as the compiler has a complete knowledge, at all points in the program, of the machine for which it is generating code). In addition, the assembly routine must be called, possibly getting parameters off a stack, while the register-aliased routine appears inline. And, as you know, the fastest call is no call at all.

The ideal use of register-aliased variables is in advanced graphics adapter drivers in which register ports must be accessed inline and in sequence with reads and writes to the screen buffer. The difference between inline code and prototypes that use C to access the ports is very noticeable.

A Sound Generator

The example presented shortly demonstrates the principles and provides a *sound( )* function that I will use as the basis for a Basic-like music interpreter. The program in Listing One (page 122) contains a general-purpose sound routine called *note( )* that can be called with a pitch and duration. The program works by controlling the 8255 timer on the system motherboard and is similar to the program in the IBM-PC ROM BIOS. The traditional Microsoft C version of the program demonstrates the principles we have developed earlier, except for *mapdev( )*.

The register-aliased version of the same program is shown in Listing Two (page 122). Note that we have defined a couple of macros at the head which convert *outp( )* from a function to a macro, and that we have created a new form of *inp( )* that is also a macro. The benefit of these macros (over the functions) is that they are almost identical in form to those used by MS C, take up much less space than function calls in the generated code, and execute inline without the costly overhead of a call and return.

In examining the code, note that the source is virtually identical to Listing One, and that the only source lines that have in fact changed are those that invoke the two interrupts and port reads. The technique used to force a variable into a particular register is a simple extension of the *reg* keyword, followed by the register to be used. To use an 8- or 16-bit section of a 32-bit register, use the 32-bit register name, followed by *short* or *char*, using *unsigned* as necessary. Variables that have the same name as the register in which they reside are called "register aliased."

It is possible, however, to overload variables, and in this demonstration I have deliberately overloaded *eax*, declaring three variables (*al*, *x*, and *y*) to be stored in *al*, another in *ax*, and another in *ah*. (Note that *ah* is specified in a slightly different manner.) None of these five *eax* components are active at the same time. If a conflict exists, the compiler spills temporaries to the stack, thus signaling us to change our usage, if possible.

Listing Three (page 122) is the assembly language produced by the compiler for the procedure *note( )*. The code demonstrates the level of integration produced by the use of register-aliased variables. The register allocation summary at the bottom shows that all of the variables in the program, including the register-aliased variables, are allocated to registers. In addition, notice that the register-coloring algorithm used by the allocator placed six variables in *eax* (this includes the use of its compo-

nents *al*, *ah*, and *ax*), in addition to using *eax* as an accumulator in three other locations. The inline assembly is integrated with the surrounding C code without a single push or pop, and does not require the use of structures or calls and returns to handle interrupts and port I/O.

a . . g[#,+,-,.,n]	the notes with optional extenders
o . . O n	octave, n = 0 . . 6
>	one octave
<	go down an octave
l . . L n	set note length n = 1 . . 64
p,P,r,R n	rests of length n = 1 . . 64
t,T n	set tempo n = 32 . . 255

Table 1: *The syntax supported by the C version of the PLAY function*

In particular, examine the block that starts at label L13. This block is the one that polls the timer until it is time to turn off the note. The block starts off with *ax* being used as an argument for the timer service routine 01AH; the *eax* register is then used by the compiler as a temporary accumulator, which is followed by *al* being used to service both input and output operations. Of course, *ecx*, *edx*, *esi*, and *edi* (or their components) are also used throughout the 12 lines of code. In fact, the only register that does not find its way into this sequence is *ebx*.

The resulting code is about one-third the size of a similar routine that uses calls. However, it executes substantially faster than the mere size difference would indicate, as most of the instructions are self-contained in the processor's registers. The issue of keeping things in registers becomes even more important with the 80486 which, even though it has a built-in cache, has been RISCed to execute many register-register instructions in a single cycle. Taking advantage of these single cycle operations becomes a key issue with generating 80486 code. In addition, future generations of 486 systems will probably increase the importance of register allocation as improved processor speeds (50 to 100 MHz) outstrip the ability of inexpensive DRAM memory to keep up with the CPU.

Conclusion

To wrap things up, Listing Four (page 122) presents the *play( )* function. The program calls *note( )*, using a case statement to parse the note being played or the command being executed. The syntax chosen was that in the IBM 2.0 Basic manual. The elements of the syntax supported are shown in Table 1.

The command is only a partial implementation of the Basic version. Because the current implementation of *note( )* polls the timer, *play( )* will not run in the background. If you plan to use this function as part of a game, I recommend that you turn the routine into a compiler that generates a list of pitches and durations that get fed to a TSR interpreter that uses the timer tick to control duration.

One final problem is the fact that C does not support strings in strings. The addition of legato or staccato is left as a reader exercise.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 122.)

Vote for your favorite feature/article.
Circle Reader Service **No. 8.**

Listing One *(Text begins on page 22.)*

```
{$V-} {$F+} {$O+}   { Written By:  Rick Gessner, 1989. }

Unit HyprText;
{-------} Interface {-----------------------------------------------}

PROCEDURE Help_Editor(FileName: String);
PROCEDURE Do_help(FileName: String; GoPage,HomePage: Word);
{-------} Implementation {--------------------------------------}
Uses Crt;

CONST HelpColor        : Array[False..true] of Byte =
                             ( Black*16+White,     { for normal text}
                               Magenta*16+Yellow); { for hot-link text.}
      NormalColor      : Byte = Black*16+White;   { draw screen info.}
      BoldColor        : Byte = White*16+Black-Blink; { for select bar.}
      Header           : String[50] = ' HyperText System [1.0] ';
      MaxLinesPerPage  = 15;
      MaxLineWidth     = 57;

      PGUP    = 'I';      PGDN    = 'Q';      UpArrow = 'H'; {Edit keys}
      DnArrow = 'P';      LArrow  = 'K';      RArrow  = 'M';
      ESC     = #27;      HomeKey = 'G';      EndKey  = 'O';
      RETURN  = ^M;       BkSpc   = #8;       NULL    = #0;
      Tab     = #9;       F2      = '<';      DelKey  = 'S';
Type  HelpRecord = Record  {The main structure for our hypertext files}
          HelpLines : Array[1..MaxLinesPerPage] of String[100];
     end; {String length MUST be > than MaxLineWidth to store hot-links!}
Var   HelpRec    : HelpRecord;
      HelpFile   : File of HelpRecord;
      Alt,Ctrl,CommandKey : Boolean;
{-----------------------------------------------------------------}
FUNCTION Make_String(Ch   : Char; Size : Integer) : String;
Var S: string;
Begin
    S[0] := Chr(Size);             { Set length byte = SIZE.          }
    FillChar(S[1],Size,Ch);        { Fill the string with chr(CH).    }
    Make_String:= S;               { and return the string as function}
end; {Make String }                { value.                           }
{-----------------------------------------------------------------}
PROCEDURE Draw_Box(topx,topy,botx,boty: Byte; Color,Width: byte);
Type BoxPos = (TopL,TopR,BotL,BotR,Top,Bot,LSide,RSide);
Var Y       : Integer;
Const  Boxchar : Array[1..2,TopL..RSide] of char =
   (( 'Z','?','@','Y','D','D','3','3'),  {ASCII chars, single line box }
    ( 'I',';','H','<','M','M',':',':')); {ASCII chars, double line box }
Begin
  TextAttr:=Color;
  If Not (Width in [1,2]) then Width:=1; { Sure width value is OK? }
  Gotoxy(TopX,TopY);    { First, draw the top line of the box...}
  Write( BoxChar[Width,TopL]+Make_String(BoxChar[width,top],
                                    BotX-TopX-1)+BoxChar[Width,TopR]);
  For Y:=TopY+1 to BotY-1 do
  Begin                 { Second, draw the middle lines of the box...}
        Gotoxy(TopX,Y);
        Write( BoxChar[Width,LSide],BoxChar[Width,RSide]:BotX-TopX);
  end;
  GotoXY(TopX,BotY);    { Third, draw the bottom line of the box. }
  Write( BoxChar[Width,BotL]+Make_String(BoxChar[width,top],BotX-TopX-1)+
                                                    BoxChar[Width,BotR])
end; {Draw Box}
{-----------------------------------------------------------------}
FUNCTION Read_KeyBoard: Char;   {Routine to get keystrokes from user}
Const  CtrlMask  = $04;
       AltMask   = $08;
Var    KBDFlag   : Byte Absolute $0040:$0017;
Begin
 Read_KeyBoard:=ReadKey;
 CommandKey :=((KBDFlag AND AltMask)<>0) or ((KBDFlag AND CtrlMask)<>0);
 ALT  :=(KBDFlag AND AltMask)<>0; CTRL := (KBDFlag AND CtrlMask)<>0;
 If KeyPressed Then
 Begin
      Read_Keyboard := ReadKey; {Just in case user pressed modified key}
         CommandKey := True;
  end;
end; {Read_Keyboard}
{-----------------------------------------------------------------}
PROCEDURE Show_HelpLine(X,Y,StartBold,EndBold: Integer; Var Line: String);
  Var I,J: Integer;
      PROCEDURE Write_Char(Ch: Char);
      Begin
           If Ord(Ch)>127 then Ch:=Chr(Ord(Ch)-128); {Clear high bit}
           If Ord(Ch)>27 then Write(Ch) else Inc(i);
      end;
  Begin
      TextAttr:=HelpColor[False];
      Window(X,Y,59,Y); ClrEOL; Window(1,1,80,25); {Prepare for output}
      Gotoxy(X,Y); I:=1;
      While I<=Length(Line) do                {Do each char in line}
      Begin
           TextAttr:=HelpColor[Ord(Line[i])>128];    {Set proper color}
           If I in [StartBold..EndBold] then TextAttr:=BoldColor;
           Write_Char(Line[i]);
           Inc(i);
      end;
 end; {Show helpline}
{-----------------------------------------------------------------}
 PROCEDURE Show_Help_Page(X,Y: Integer; Var HelpRec: HelpRecord);
 Var I: Integer;
 Begin
    Window(X+1,Y+1,X+56,Y+MaxLinesPerPage+1); ClrScr; Window(1,1,80,25);
    For I:=1 to MaxLinesPerPage do
        Show_HelpLine(X,Y+I,0,0,HelpRec.HelpLines[I]);
 end; {Show help page}
{-----------------------------------------------------------------}
 FUNCTION Determine_Actual_Line_Pos(Var Line: String; LinePos:
                                                    Integer): Integer;
 Var I,J: Integer;  {Convert visual edit column to char. position,}
 Begin              {by skipping over embedded hot links.}
      I:=0; J:=1;
      While (J<=Length(Line)) and (I<>LinePos) do
      Begin
           Inc(i) else Inc(j,2);
         Inc(j);
      end;
      Determine_Actual_Line_Pos:=J;
end; {Determine actual line pos}

{-----------------------------------------------------------------}
FUNCTION Link_Count(Var Line: String): Integer;
Var I,Count: Integer;  { Returns 2*#nulls in line, used to convert }
Begin                  { from actual byte pos. to visual byte pos., }
     Count:=0;         { during data input. }
     For I:=1 to Length(Line) do
         If Line[i]=Null then Inc(Count,2);
     Link_Count:=Count;
end; {Link count}
{-----------------------------------------------------------------}
FUNCTION Input_HelpPage(X,Y: Byte; Var AHelpRec: HelpRecord): Char;
Var   Ch         : Char;   { Main editing routine in this system.  }
      PageNum    : Byte;   { It is really just a page-oriented line }
      I,J,                 { editor knows how to jump over 2-byte }
      LinePos,             { hot-links. }
      RealLinePos,         { If you add editing options, take }
      LineNum    : Integer;{ the embedded hot-links into account! }

      PROCEDURE Delete_Linked_Char(Var Line: String; LinePos: Integer);
      Var I,J: Integer;
      Begin
          LinePos:=Pred(Determine_Actual_Line_Pos(Line,LinePos));
          f Ord(Line[LinePos])>127 then  {Were on a linked item}
          egin
          I:=LinePos;
          While ((Ord(Line[I-1])>127) and (I>1)) do Dec(i);
          J:=LinePos;                          {Next find end of link}
           While ((Ord(Line[J+1])>127) and (I<Length(Line))) do Inc(J);
              Delete(Line,LinePos, {Delete all of item + link if necc.}
                     1+(2*Ord(J=I)));
          end;
      end; {Delete linked char}
Begin
     Show_Help_Page(X,Y,AHelpRec);  {Display this page }
     LinePos:=1; RealLinePos:=1;    {Now do a little init stuff.}
     LineNum:=1;
     With AHelpRec do               {Now enter main edit loop...}
     Repeat
           Show_HelpLine(X,Y+LineNum,0,0,HelpLines[LineNum]);
           Gotoxy(X+LinePos-1,Y+LineNum);
           Repeat Ch:=Read_KeyBoard Until Ch <> Null;
           If CommandKey then
              Case Ch of
               ^Y     : If RealLinePos<=Length(HelpLines[LineNum]) then
                        Begin    { ^Y = Delete to end of line. }
                          If (RealLinePos=1) then HelpLines[LineNum]:=''
                          else
                          Begin
                            While HelpLines[LineNum,RealLinePos]<>Null do
                            Delete(HelpLines[LineNum],RealLinePos,1);
                           If HelpLines[LineNum,RealLinePos]=Null then
                            Delete(HelpLines[LineNum],RealLinePos+2,255)
                           end
                        end;
               F2     : Begin { F2 = Add/Remove hot-link.}
                          J:=RealLinePos;
                          While (j>0) and (HelpLines[LineNum,j]<>' ')
                                                          do Dec(j);
                          Inc(j);
                          If Ord(HelpLines[Linenum,j]) in [28..127] then
                          Repeat {Now get a valid page # to jump to...}
                                   Gotoxy(3,24); Write('Link Page: ');
                                   Readln(PageNum);
                                   Gotoxy(3,24); ClrEOL;
                               Until (PageNum>0) and (PageNum<256);
                               While (HelpLines[LineNum,j]<>' ') and
                                     (j<=Length(HelpLines[LineNum])) and
                                    (HelpLines[LineNum,j]<>Null) do
                               Begin
                                 HelpLines[LineNum,j]:=Chr(Ord(HelpLines
                                                      [LineNum,j])+128);
                                    Inc(j);
                               end;
                        If Ord(HelpLines[LineNum,J-1]) in [28..127] then
                        Delete(HelpLines[LineNum],J,2) else
                        Insert(Null+Chr(PageNum),HelpLines[LineNum],j);
                        end;
               LArrow : If RealLinePos>1 then {Move cursor left 1 char}
                      Begin
                        Dec(linePos);
                        RealLinePos:=Pred(Determine_Actual_Line_Pos
                                      (HelpLines[LineNum],LinePos));
                      end;
               RArrow : If RealLinePos<=Length(HelpLines[LineNum]) then
                        Begin               {Move cursor right 1 char}
                          Inc(LinePos);
                          If RealLinePos<Length(HelpLines[LineNum]) then
                                  Inc (RealLinePos,
                          1+Ord(HelpLines[LineNum,RealLinePos+1]=Null)*2)
                                  else Inc(realLinePos)
                        end;
               DnArrow: If LineNum<MaxLinesPerPage then
                      Begin                 {Move down 1 line.}
                        Inc(LineNum);
                        If LinePos<=Length(HelpLines[LineNum]) then
                          RealLinePos:=Pred(Determine_Actual_Line_Pos
                                      (HelpLines[LineNum],LinePos))
                        else
                      Begin
                        RealLinePos:=Succ(Length(HelpLines[LineNum]));
                        LinePos:=RealLinePos-Link_Count(HelpLines[LineNum]);
                      end;
                    end;
```

(Listing continued on page 88)

Listing One *(Listing continued, text begins on page 22.)*

```
                UpArrow: If LineNum>1 then      {Move up 1 line.}
                     Begin
                        Dec(LineNum);
                        If LinePos<=Length(HelpLines[LineNum]) then
                          RealLinePos:=Pred(Determine_Actual_Line_Pos
                                            (HelpLines[LineNum],LinePos))
                        else
                     Begin
                      RealLinePos:=Succ(Length(HelpLines[LineNum]));
                      LinePos:=RealLinePos-Link_Count(HelpLines[LineNum]);
                     end;
                   end;
                HomeKey: Begin                  {Move to 1 char. in line.}
                              LinePos:=1;
                              RealLinePos:=LinePos;
                         end;
                EndKey : Begin                  {Move to end of line.}
                    RealLinePos:=Succ(Length(HelpLines[LineNum]));
                    LinePos:=RealLinePos-Link_Count(HelpLines[LineNum]);
                    end;
              DelKey : If (RealLinePos<=Length(HelpLines[LineNum])) then
                   Begin                        {Delete a character.}
                    If (HelpLines[LineNum,RealLinePos]) in [' '..'}'']
                     then Delete(HelpLines[LineNum],RealLinePos,1) else
                    Delete_Linked_Char(HelpLines[LineNum],LinePos);
                    RealLinePos:=Pred(Determine_Actual_Line_Pos
                                      (HelpLines[LineNum],LinePos));
                    end;
                 end else
            Case Ch of
            Return: If LineNum<MaxLinesPerPage then  {Move down 1 line.}
                         Begin
                              Inc(LineNum); LinePos:=1; RealLinePos:=1;
                         end;
                Tab : Begin     {Tab right 10 chars.}
                 If RealLinePos+10<=Length(HelpLines[LineNum])+1 then
                              Inc(RealLinePos,10) else
                  RealLinePos:=Length(HelpLines[LineNum])+1;
                  LinePos:=RealLinePos-Link_Count(HelpLines[LineNum]);
                       end;
            BkSpc  : If RealLinePos>1 then  {Backspace over prev. char.}
                Begin
                  If HelpLines[LineNum,RealLinePos-1] in [' '..'}'] then
                  Begin
                       Delete(HelpLines[LineNum],RealLinePos-1,1);
                       Dec(RealLinePos);
                       Dec(LinePos)
                  end else
                  Begin
                       Delete_Linked_Char(HelpLines[LineNum],LinePos-1);
                       Dec(LinePos);
                       RealLinePos:=Pred(Determine_Actual_Line_
                                         Pos(HelpLines[LineNum],LinePos));
                  end;
                end;
           ' '..'}' : If Length(HelpLines[LineNum])<MaxLineWidth then
                   Begin     {Insert a valid Ascii char.}
                     If (Ord(HelpLines[LIneNum,RealLinePos])>127) and
                        (RealLinePos<=Length(HelpLines[Linenum])) then
                        Ch:=Chr(Ord(Ch)+128);
                     Insert(Ch,HelpLines[LineNum],RealLinePos);
                     Inc(RealLinePos);
                     Inc(LinePos); Ch:=#255;
                    end;
             end;
   Until CH in [ESC,PGUp,PgDn]; {ESC=Quit;PGUp=Prev page;PgDn=Next Page.}
   Input_HelpPage:=Ch;
  end; {Input helppage}
{-------------------------------------------------------------------}
  FUNCTION Read_Helprec(Var AHelpRec:HelpRecord; RecNum:Integer):Integer;
  Var I : Integer;
  Begin
   FillChar(AHelprec,SizeOf(AHelprec),0); {$I-} {Hyperdata file read rec}
   If FileSize(HelpFile)<RecNum then exit;      {routine. Includes just }
   Seek(helpfile,RecNum-1);                     {enough error checking  }
   Read(helpfile,AHelpRec);                     {to be considered safe. }
   Read_HelpRec:=IOResult; {$I+}
  end; {Read helprec}
{-------------------------------------------------------------------}
 FUNCTION Write_HelpRec(Var AHelpRec:HelpRecord; RecNum:Integer):Integer;
  Begin {$I-}
    Seek(helpfile,RecNum-1);          {Hyperdata file write rec routine.}
    Write(helpfile,AHelpRec); {$I+}  {This routine also contains just  }
    Write_HelpRec:=IOresult;          {enough error checking to be      }
  end; {Write helprec}                {considered safe.                 }
{-------------------------------------------------------------------}
  FUNCTION Open_HelpFile(FileName: String): Integer;
  Var result: Integer;
  Begin
       Assign(HelpFile,FileName); {$I-}  {Opens hyperdata file specified}
       Reset(HelpFile);                  {as "FileName".  If the file   }
       result:=IOResult;                 {doesnt exist, then it will be }
       If Result=2 then                  {created.                      }
       Begin                             {Error checking is limited, but}
            ReWrite(HelpFile);           {enough to be safe.            }
            Result:=IOResult;
       end;
       Open_HelpFile:=Result;
  end; {open helpfile}
{-------------------------------------------------------------------}
 PROCEDURE Help_Editor(FileName: String);
 Const  HelpMsgs  = 13;
        HelpData  : Array[1..HelpMsgs] of String[17] =
              ( 'Editing Keys: ',          '-------------',
               'F2   : Link (+/-)',       '^Y   : Del EOLine',
               'Bkspc: Del left',         'Del  : Del char',
               Chr(10)+'Movement keys: ', '-------------',
               Chr(24)+Chr(25)+Chr(27)+Chr(26)+', '+Chr(17)+Chr(217)+',',
               'Tab, Home, End',          'PgUp : Prev page',
               'PgDn : Next page',        Chr(10)+'ESC to quit.');
 Var I,HelpRecNum: Integer;
     AHelpRec    : HelpRecord;
     Ch          : Char;
     Result      : Integer;
 Begin
      Result:=Open_HelpFile(FileName);     {Open the specified file.}
      If Result=0 then                     {Continue only if no error.}
      Begin
           TextAttr := NormalColor;
           Draw_Box(1,3,80,23,NormalColor,1);
           Draw_Box(2,4,60,22,NormalColor,2);
           Gotoxy(61,4);
           For I:=1 to HelpMsgs do
           Begin
                Gotoxy(62,WhereY+1); Write(HelpData[i]);
           end;
           HelpRecNum:=1;
           Gotoxy(40-(Length(Header) div 2),3); Writeln(Header);
           Gotoxy(4,2); Writeln('File: ',FileName);
           Repeat
                Gotoxy(4,4); Writeln(' Reading  ');
                Result:=Read_HelpRec(AHelpRec,HelpRecNum);
                Gotoxy(4,4); Writeln('Page: ',HelpRecNum:3);
                 Ch:=Input_HelpPage(3,4,AHelpRec);
                Result:=Write_HelpRec(AHelpRec,HelpRecNum);
                Gotoxy(4,4); Writeln(' Writing  ');
                Case Ch of
                     PgUp : If helpRecNum>1 then Dec(HelpRecNum);
                     PgDn : If HelpRecNum < 255 then Inc(HelpRecNum);
                end;
           Until Ch=ESC;
      end else  {Report the opening error...}
   Writeln('ERROR: ',Result,' opening ',FileName,'. Unable to continue.');
   {$I-} Close(HelpFile); Result:=IOresult; {$I+}
 end; {Help editor}
{-------------------------------------------------------------------}
 FUNCTION Find_Next_Link( Var X,Y: Integer; EndX,EndY: Integer;
                          Var AHelpRec: HelpRecord): Boolean;
 Var OrigX,OrigY,Col,                 {Recursive routine used to find a }
     Row,StartCol,StopCol: Integer;   {hot-link on the page after the   }
 Begin                                {current page position (X,Y).     }
      Find_Next_Link:=False;
      {First, search from current pos to end of page...}
      For Row:=Y to EndY do
      Begin
           If Row<>Y then StartCol:=1 else StartCol:=X;
           If Row<>EndY then StopCol:=Length(AhelpRec.HelpLines[Row])
              else StopCol:=EndX;
           If AhelpRec.HelpLines[Row]<>'' then
           For Col:=StartCol to StopCol do
                If (AHelpRec.HelpLines[Row,Col]=Null) then
                Begin
                     Find_Next_Link:=True;
                     X:=Col; Y:=Row;
                     Exit;  {make a quick getaway!}
                end;
      end;
      {ok, search from top of page to the startpos}
      If X+Y>2 then
      Begin
           Col:=1; Row:=1;
           If Find_Next_link(Col,Row,Pred(X),Y,AHelpRec) then
           Begin
               X:=Col; Y:=Row; Find_Next_Link:=true;
           end
      end;
 end; {find next link}
{-------------------------------------------------------------------}
 FUNCTION Find_Prev_Link( Var X,Y: Integer; EndX,EndY: Integer;
                          Var AHelpRec: HelpRecord): Boolean;
 Var OrigX,OrigY,Col,                 {Recursive routine used to find a }
     Row,StartCol,StopCol: Integer;   {hot-link on the page prev. to the}
 Begin                                {current page pos. (X,Y).         }
      Find_Prev_Link:=False;
      {First, search from current pos to top of page...}
      For Row:=Y downto 1 do
      Begin
           StopCol:=1;
           If Row<>Y then StartCol:=Length(AhelpRec.HelpLines[Row])
              else StartCol:=X;
           If AhelpRec.HelpLines[Row]<>'' then
           For Col:=StartCol downto StopCol do
                If (AHelpRec.HelpLines[Row,Col]=Null) then
                Begin
                     Find_Prev_Link:=True;
                     X:=Col; Y:=Row;
                     Exit;  {make a quick getaway!}
                end;
      end;
      {ok, search from bottom of page to the startpos}
      If X+Y>2 then
      Begin
           Row:=MaxLinesPerPage;
           Col:=Length(AHelpRec.HelpLines[Row]);
           If Find_Prev_link(Col,Row,Succ(X),Y,AHelpRec) then
           Begin
               X:=Col; Y:=Row; Find_Prev_Link:=true;
           end
      end;
 end; {find prev link}
{-------------------------------------------------------------------}
 PROCEDURE Do_Help(FileName: String; GoPage,HomePage: Word);
 Const XPos = 10;
     YPos = 5;
     Color    : Byte = Black*16+White;   {This is the hypertext engine.}
     MaxStackSize = 25;                  {This routine is used to read }
                                         {and navigate through a data  }
 Type  StackRec = Record                 {file, specfied as "FILENAME".}
             Page : Byte;                {GoPage specifies the starting}
             Row,                        {page to display, and HomePage}
```

(Listing continued on page 90)

Listing One *(Listing continued, text begins on page 22.)*

```
            Col  : Integer;              {is used to specify an main   }
       end;                              {index (or home) page.        }
 Var   Result  : Integer;
       Stack   : Array[0..MaxStackSize] of StackRec;
       AHelpRec: HelpRecord;
       Ch      : CHar;
       StackLvl: Byte;
       StartCol: Integer;
       Linked,
       Load    : Boolean;
       FUNCTION Pop_Stack: Byte;  {Pop the top page info (Stack) record}
       Begin
            If StackLvl>1 then
            Begin
                 Dec(StackLvl);
                 Load:=True;
            end;
            Pop_Stack:=StackLvl;
       end; {pop stack}
       FUNCTION Push_Stack(PageNum: Byte): Byte;
       Begin                          {Push a page info (stack) record.}
            Inc(StackLvl);
            Stack[StackLvl].Page:=PageNum;
            Stack[StackLvl].Col:=1;
            Stack[StackLvl].Row:=1;
            Push_Stack:=StackLvl;
       end; {push stack}
 Begin
      If GoPage=0 then GoPage:=1;      {Make sure GoPage is valid.}
      Result:=Open_HelpFile(FileName);
      If Result=0 then
      Begin
        Load:=true;
        TextAttr :=Color;
        Draw_Box(Xpos,YPos,XPos+59,YPos+MaxLinesPerPage+2,NormalColor,2);
        FillChar(Stack,SizeOf(Stack),0);
        StackLvl := 0;
        If HomePage in [1..255] then StackLvl:=Push_Stack(HomePage);
        If (GoPage in [1..255]) and (GoPage<>HomePage) then
                                          StackLvl:=Push_Stack(GoPage);
        GotoXY(XPos+29-(Length(Header) div 2),YPos);
        Writeln(Header);
        Repeat
                 With Stack[StackLvl] do
                 Begin
                    If Load then  {System needs new hyperdata file page.}
                    Begin
                       Result:=Read_HelpRec(AHelpRec,Page);
                       Show_Help_Page(XPos+1,YPos,AHelpRec);
                       Gotoxy(XPos+51,YPos+MaxLinesPerPage+2);
                       If StackLvl>1 then Write('Esc,PgUp') else
                                                  Write('Esc=Quit');
                       Linked:=Find_Next_Link(Col,Row,80,MaxLinesPerPage,
                                                         AHelpRec);
                       Load:=False;
                    end;
                    If Linked then {We have a hot-link to show, so do it.}
                    Begin
                       StartCol := Col;
                       While Ord(AHelprec.HelpLines[Row,StartCol-1])>127
                                                      do Dec(StartCol);
                       Show_HelpLine(XPos+1,YPos+Row,StartCol,Pred(Col),
                                              AHelpRec.HelpLines[Row]);
                    end;
                 Repeat Ch:=Read_KeyBoard until Ch<>Null;
              Show_HelpLine(XPos+1,YPos+Row,0,0,AHelpRec.HelpLines[Row]);
              Case Ch of   {Now handle navigation...}
                           RArrow,
                           Tab    : Begin
                                      Inc(Col);
                                      Linked:=Find_Next_Link(Col,Row,80,
                                              MaxLinesPerPage,AHelpRec);
                                    end;
                           Return : If Linked then
                              Begin
                                Load:=true;
                                If (StackLvl>1) and
                                (Stack[StackLvl-1].Page=
                                 Ord(AHelpRec.HelpLines[Row,Col+1])) then
                                StackLvl:=Pop_Stack else
                                StackLvl:=Push_Stack(Ord(AHelpRec.HelpLines
                                                      [Row,Col+1]));
                                 end;
                       LArrow : Begin
                         Dec(Col);
                         Linked:=Find_Prev_Link(Col,Row,1,1,AHelpRec);
                         end;
                       DnArrow: Begin
                         Col:=1;
                         If Row<MaxLinesPerPage then Inc(Row) else Row:=1;
                         Linked:=Find_Next_Link(Col,Row,80,
                                              MaxLinesPerPage,AHelprec);
                         end;
                       UpArrow: Begin
                         If Row>1 then Dec(Row) else Row:=MaxLinesPerPage;
                         If Col<Length(AHelpRec.HelpLines[Row])
                                                    then Inc(Col);
                         Linked:=Find_Prev_Link(Col,Row,1,1,Ahelprec);
                         end;
                       PgUp   : StackLvl:=Pop_Stack;
                       PgDn   : Begin  {Let programmer set this!} end;
                 end;
              end;
         Until Ch=ESC;
  end else
   Writeln('ERROR: ',Result,' opening ',FileName,'. Unable to continue.');
  {$I-} Close(HelpFile); result:=IOResult; {$I+}
  end; {do help}
 {-------------------------------------------------------------------}
  Begin {No init code required}
  end.
```

End Listing

Listing One *(Text begins on page 34.)*

```
/*-----------------------------------------
  Demonstrates basic principles of hypertext
  document construction and management.
  (c) Copyright 1989 Todd King
        All Rights Reserved
  Written: Todd King
-----------------------------------------*/

#include <stdio.h>
#include <string.h>
#include <conio.h>
#include <ctype.h>
#include "hyper_d.h"

main()
{
  int n;

  clrscr();
  printf("(c) Copyright 1989 Todd King. All Rights Reserved\n");
  sleep(2);

  if((Hyper_fptr = fopen("HYPER.TXT", "r")) == NULL)
  {
    perror("fopen");
    exit(1);
  }

  build_hyperbase();

  n = 0;
  while((n = enter_text(Hyperbase[n].tag)) != -1) ;

  clrscr();
  fclose(Hyper_fptr);
}

/*-----------------------------------------
  Builds the global index data base for the
  hypertext document.
  Written: Todd King
-----------------------------------------*/
build_hyperbase()
{
  char buffer[MAX_HYPER_LINE];
  HYPERITEM *hitem;
   int n;

  for(;;)
  {
    if(fgets(buffer, sizeof(buffer), Hyper_fptr) == NULL)
    {
      break;
    }
    switch(buffer[0])
    {
      case '\f':
        if((hitem = make_item()) == NULL)
        {
          fprintf(stderr, "No more room in the Hyperbase\n");
          exit(0);
        }
        fgets(buffer, sizeof(buffer), Hyper_fptr);
        n = strlen(buffer);
        if(buffer[n - 1] == '\n') buffer[n - 1] = '\0';
        set_tag(hitem, buffer);
        set_position(hitem, ftell(Hyper_fptr));
        break;
    }
  }
  return(Hyper_cnt);
}

/*-----------------------------------------
  Enters the hypertext document at a specific
  tag location. It then allows navigation within
  the document.
  Written: Todd King
-----------------------------------------*/
enter_text(tag)
char tag[];
{
  int c, n;
  HYPERITEM *hitem;
  char *bptr;
  char *pptr;
  char *tptr;
  int i;
  char buffer[MAX_HYPER_LINE];
  int line_put;
  int line_cnt = 3;
  int col_cnt = 0;
  int ref_cnt = 0;
  int hidx;
  HYPER_REF hyper_ref[MAX_HYPER];

  hitem = find_item(tag);

  if(hitem == NULL)
  {
     fprintf(stderr,
      "No hypertext subject by the name of '%s' exists\n", tag);
    return(-1);
  }
  hidx = hyper_idx(tag);
  clrscr();
  fseek(Hyper_fptr, hitem->position, SEEK_SET);
  bptr = fgets(buffer, sizeof(buffer), Hyper_fptr);
  textcolor(LIGHTGRAY);
  gotoxy(1,24);
  cputs("ESCAPE: to exit; UP ARROW, DOWN ARROW: to navigate");
  gotoxy(1,1);
  cputs(tag); cputs("\r\n"); cputs("\r\n");
  while(bptr != NULL)   /* For all lines in the current hypercard */
  {
    col_cnt = 1;
    for(i = 0; i < Hyper_cnt; i++)
    {
     if((pptr = strstr(bptr, Hyperbase[i].tag)) != NULL)
     {
       if( i == hidx) continue; /* no self-referencing */
       if(pptr != bptr)
       {
         if(!ispunct(*(pptr - 1)) && !isspace(*(pptr - 1))) continue;
       }
       tptr = pptr + strlen(Hyperbase[i].tag);
       if(ispunct(*tptr) || isspace(*tptr) || *tptr == '\0')
       {
         /* Deliniator */
       } else {          /* No good */
         continue;
       }
 /* If we reach here we've found a genuine tag */
        pptr[0] ='\0';
        col_cnt += strlen(bptr);
        hyper_ref[ref_cnt].line = line_cnt;
        hyper_ref[ref_cnt].column = col_cnt;
        hyper_ref[ref_cnt].tag = Hyperbase[i].tag;
        cputs(bptr);
        textcolor(YELLOW);
        cputs(Hyperbase[i].tag);
        textcolor(LIGHTGRAY);
        bptr = pptr + strlen(Hyperbase[i].tag);
        col_cnt += strlen(Hyperbase[i].tag);
        ref_cnt++;
      }
    }
    cputs(bptr);
    cprintf("\r");
    if(line_cnt >= 23) {  /* What to do at the end of a screen */
       break;
    }
    bptr = fgets(buffer, sizeof(buffer), Hyper_fptr);
    if(buffer[0] == '\f') bptr = NULL;
    line_cnt++;
    col_cnt = 0;
  }
  if((n = nav_ref(hyper_ref, ref_cnt)) >= 0) return(n);

}

/*-----------------------------------------
```

```
  The function that performs the actual
  navigation within a hypertext document.
  Written: Todd King
-------------------------------------------*/
nav_ref(hyper_ptr, max_ref)
HYPER_REF hyper_ptr[];
int max_ref;
{
  int advance = 0;
  int selected = -1;
  int cur_ref = 0;

  for(;;)
  {
    cur_ref += advance;
    if(cur_ref < 0) cur_ref = max_ref - 1;
    if(cur_ref >= max_ref) cur_ref = 0;
    advance = 0;
    gotoxy(hyper_ptr[cur_ref].column, hyper_ptr[cur_ref].line);
    textbackground(LIGHTGRAY);
    textcolor(BLACK);
    cputs(hyper_ptr[cur_ref].tag);
    switch(getch())
    {
      case 0:
        switch(getch())
        {
          case DOWN_ARROW:
            advance = 1;
            break;
          case UP_ARROW:
            advance = -1;
            break;
        }
        break;
      case '\r':
      case '\n':
        selected = cur_ref;

        break;
       case ESC:
        textbackground(BLACK);
        textcolor(LIGHTGRAY);
        return(-1);
    }
    gotoxy(hyper_ptr[cur_ref].column, hyper_ptr[cur_ref].line);
    textcolor(YELLOW);
    textbackground(BLACK);
    cputs(hyper_ptr[cur_ref].tag);
    if(selected != -1) return(hyper_idx(hyper_ptr[selected].tag));
  }
}

/*-------------------------------------------
  Determines the index of a hypertext tag
  within the global database.
  Written: Todd King
-------------------------------------------*/
hyper_idx(tag_str)
char tag_str[];
{
  int i;
  for(i = 0; i < Hyper_cnt; i++) {
    if(strcmp(tag_str, Hyperbase[i].tag) == 0) return(i);
  }
  return(-1);
}

/*-------------------------------------------
  Locates an item in the global database
  with the tag as a key. Returns a pointer
  to the entry or NULL if one does not exist.
  Written: Todd King
-------------------------------------------*/
HYPERITEM *find_item(tag)
char tag[];
{
  HYPERITEM *hitem;
  int i;

  for(i = 0; i < Hyper_cnt; i++)
  {
    hitem = &Hyperbase[i];
    if(strcmp(tag, hitem->tag) == 0) return(hitem);
  }
  return(NULL);
}

/*-------------------------------------------
  Sets the tag portion of a database entry to
  the contents in a passed string
  Written: Todd King
-------------------------------------------*/
set_tag(hitem, buffer)
HYPERITEM *hitem;
char buffer[];
{
  char *malloc();

  hitem->tag = malloc(strlen(buffer) + 1);
  if(hitem->tag == NULL)
  {
    perror("malloc");
    exit(2);
  }
  strcpy(hitem->tag, buffer);
}
```

(Listing continued on page 94)

Listing One *(Listing continued, text begins on page 34.)*

```
/*-----------------------------------------
  Makes (extracts) a new database entry item
  from the item pool.
  Written: Todd King
-----------------------------------------*/
HYPERITEM *make_item()
{
  if(Hyper_cnt >= MAX_HYPER) return(NULL);
  return(&Hyperbase[Hyper_cnt++]);
}
```

End Listing One

Listing Two

```
#define UP_ARROW        72
#define DOWN_ARROW      80
#define ESC             27

typedef struct {
  char *tag;
  int position;
} HYPERITEM;

typedef struct {
    int line;
    int column;
    char *tag;
  } HYPER_REF;

#define MAX_HYPER       1024
HYPERITEM Hyperbase[MAX_HYPER];
int Hyper_cnt = 0;

FILE *Hyper_fptr;

HYPERITEM *make_item();
HYPERITEM *find_item();
#define set_position(h, p)  h->position = p

#define TRUE    1
#define FALSE   0

#define MAX_HYPER_LINE  80
```

End Listing Two

Listing Three

```
.....................................................................
Function Flow Diagram

 main()       build_hyperbase()             make_item()

                                              set_tag()

                                            set_position()

                enter_text()                 find_item()

                                            hyper_idx()

                                              nav_ref()
.....................................................................
main()

This is an application that displays a hypertext document.
It automatically creates a list of hyper-items (or cards)
that are in the hypertext document. Then it creates the
appropriate links so that you can navigate to any item
that is referenced by any other item. It uses the contents
of the file "HYPER.TXT" as the hypertext document and begins
at the first item in the document.


Function Flow Diagram
.....................................................................
build_hyperbase()

build_hyperbase()

This function builds the global database for the hypertext
document. It scans the entire document and assembles a list
of all cards in the document and stores their location
within the file and the tag, which the card is to be known
by. This database is stored in the global variable
"Hyperbase".


Function Flow Diagram
.....................................................................
enter_text()

enter_text(tag)
char tag[];

Enters the hypertext document at a specific
tag location. The tag (a string) is passed in the variable
first variable called "tag". It then allows navigation
within the document.


Function Flow Diagram
.....................................................................
nav_ref()

nav_ref(cur_ref, hyper_ptr, max_ref)
int cur_ref;
HYPER_REF hyper_ptr[];
int max_ref;
```

```
This function performs the actual navigation within a
single card. It returns the index of the hyper-item within
the card which was selected. A special code (-1) is returned
if a request to exit is entered. "cur_ref" is the index of
the current card, "hyper_ptr" is a pointer to a structure
containing the list of references in the card and "max_ref
is the number of references in "hyper_ptr".


Function Flow Diagram
.....................................................................
hyper_idx()

hyper_idx(tag_str)
char tag_str();

Returns the index of the tag "tag_str". The global
database ("Hyperbase") is searched for the existence of
the tag.


Function Flow Diagram
.....................................................................
find_item()

HYPERITEM *find_item(tag)
char tag[];

Locates an item in the global database ("Hyperbase")
with the tag as a key. Returns a pointer
to the entry or NULL if one does not exist.


Function Flow Diagram
.....................................................................
set_tag()

set_tag(hitem, buffer)
HYPERITEM *hitem;
char buffer[];

Sets the tag portion of the database entry pointed
to by "hitem" to the contents in the string "buffer".


Function Flow Diagram
.....................................................................
make_item()

HYPERITEM *make_item()

Makes a new database entry. Actually it extracts
the next available entry for a pool and returns a pointer
to the entry.


Function Flow Diagram
.....................................................................
set_position()

set_position(h, p)
HYPERITEM *h;
int p;

Actually a psuedofunction (created with a define) which
assigns an location of an item in a file to the item
definition.


Function Flow Diagram
.....................................................................
HYPERITEM

A structure that contains a complete description a single
item (or card) with the hypertext document. It is of the
form:

typedef struct {
  char *tag;
  int position;
} HYPERITEM;


Function Flow Diagram
.....................................................................
HYPER_REF

A structure of the form:

typedef struct {
    int line;
    int column;
    char *tag;
  } HYPER_REF;


Function Flow Diagram
.....................................................................
HYPER.TXT

The text file that contains the hypertext document. The
beginning of an item is marked by a special line. This line
is a formfeed followed by a newline. The next line after
this is considerd to be the name of the item (the item tag).
If this name appears in any other tag then navigation to
the item is allowed by selecting the tag in the item.


Function Flow Diagram
```

End Listings

Listing One *(Text begins on page 50.)*

```
/* FILETEST.HPP Written by Kevin D. Weeks Released to the Public Domain  */

#ifndef FILESPEC_HPP                        // prevent multiple #includes
#define FILESPEC_HPP

#include <stdio.h>

#define ERR     -1
// create a boolean type
typedef enum{FALSE,TRUE}     bool;

// specify attribute sizes
#define SIZE_DEVICE 2
#define SIZE_PREFIX 64
#define SIZE_NAME   9                       // the dot is part of the name
#define SIZE_SUFFIX 3

// these constants are used as flags in the condition attribute
#define FLAG_DEVICE     0x0001
#define FLAG_PREFIX     0x0002
#define FLAG_NAME       0x0004
#define FLAG_SUFFIX     0x0008
#define INCOMPLETE      0x000f
#define INVALID_CHAR    0x00f0
#define READ_ONLY       0x0100

class   File_Spec
{
    // first define the attributes
    char    device[SIZE_DEVICE + 1];        // under MS-DOS, the disk drive
    char    *prefix;                        //   "     "   , the path
    char    name[SIZE_NAME + 1];            //   "     "   , still the name
    char    suffix[SIZE_SUFFIX + 1];        //   "     "   , the extension
    char    *request;                       // pointer to response string for
                                            // get_XXXX() methods
    int     prefix_length;
    int     request_length;
    unsigned int    condition;              // current status of the object

    // and then the private methods
    bool    check_prefix(void);             // determine completeness of prefix
    bool    parse_prefix(void);             // interpret relative prefix
    bool    check_chars(char *string, unsigned int attrib_flag);
                                            // test for valid MS-DOS file chars
    void    copy(const File_Spec& original);  // copy another file spec
    bool    clear_attribute(char *attribute, unsigned int attrib_flag);
    bool    realloc(char **pointer, int *length, int new_length);

    // make class File_Spec a friend of itself
    friend  class File_Spec;

  // now the public methods
  public:
    // construct file specifications
            File_Spec(void);
            File_Spec(const char *file);
            File_Spec(const File_Spec& original);

    // destroy a file specification
            ~File_Spec(void);

    // return status of object
    unsigned int    status(void);

    // extend the language by overloading the = operator
    File_Spec   operator=(const File_Spec& original);

    // ALL get_XXXX() methods guarantee to return NUL-terminated strings
    char    *get_device(void);
    char    *get_prefix(void);
    char    *get_name(void);
    char    *get_suffix(void);
    char    *filespec(void);

    // ALL change_XXXX() methods guarantee to copy no more than SIZE_n
    // characters from the pass parameter
    bool    change_device(const char *string = NULL);
    bool    change_prefix(const char *string = NULL);
    bool    change_name(const char *string = NULL);
    bool    change_suffix(const char *string = NULL);

    // disables/enables change_XXXX()
    void    read_only(bool flag);

    // attempt to complete the file specification
    bool    complete(void);
};

#endif
```

End Listing One

Listing Two

```
/* FILESPEC.CPP Written by Kevin D. Weeks Released to the Public Domain  */

#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <ctype.h>
#include "filespec.hpp"

// this declaration instructs the compiler to NOT perform name-mangling
// on these functions.
extern "C"
{
```

```
    extern char     ll_get_drive(void);
    extern int      ll_get_cwd(int, char *);
    extern unsigned int ll_write(int, unsigned int, const void *);
}

// these constants are states used in parsing the file string
#define DEVICE  1
#define PREFIX  2
#define NAME    3
#define SUFFIX  4

// a macro to return out of memory errors
#define MEM_ERR(length) { errno = ENOMEM; length = 0; return NULL; }

extern volatile int errno;

/*  This final File_Spec constructor is passed a character string that it
    attempts to parse into its various components. Parsing is done with
    a finite state machine that begins at the end of the string and backs
    up to the beginning, changing state as it encounters the element delimiters
    ':', '.', and '\'. Once the string has been parsed the components are
    checked for completeness and for the validity of the file characters.
    In order to correctly interpret a string with a prefix but no name the
    string must end with a '\'. Also note that any individual component
    device, name, etc.) that is too long is truncated to a legal length.
*/
File_Spec::File_Spec(const char *file)
{
    char    *tmp_file;                  // local copy of tmp_file
    char    *tmp_prefix;                // temporary string for the prefix
    int     pos;                        // current position in tmp_file string
    int     state;                      // current state
    int     i;                          // trash variable
    // initialize everything that needs initializing
    prefix = NULL;
    prefix_length = 0;
    request_length = 0;
    condition = 0;
    device[0] = device[SIZE_DEVICE] = '\0';
    name[0] = name[SIZE_NAME] = '\0';
    suffix[0] = suffix[SIZE_SUFFIX] = '\0';
    errno = 0;
    if (file == NULL || *file == '\0')
    {
        condition = INCOMPLETE;
        return;
    }
    if ((tmp_file = new char[strlen(file) + 1]) == NULL)
    {
        errno = ENOMEM;
        return;
    }
    strcpy(tmp_file,file);
    if ((tmp_prefix = new char[SIZE_PREFIX + 1]) == NULL)
    {
        errno = ENOMEM;
        return;
    }
    tmp_prefix[0] = tmp_prefix[SIZE_PREFIX] = '\0';
    pos = strlen(tmp_file) - 1;             // set pos to last character
    // this while loop is the finite state machine mentioned above. note
    // that the strncpy() calls copy everthing from just beyond the
    // character that satisfies the case, and then the tmp_file is truncated
    // at that point with a '\0'.
    state = SUFFIX;
    do
    {
        switch (tmp_file[pos])
        {
            case '.':
                // a dot only counts in the SUFFIX state
                if (state == SUFFIX)
                {
                    strncpy(suffix,&tmp_file[pos + 1],SIZE_SUFFIX);
                    tmp_file[pos + 1] = '\0';
                    state = NAME;
                }
                else
                    if (state == NAME)
                        // this means we've got two or more dots in a name
                        // which is illegal. flag it as an invalid char
                        condition |= FLAG_NAME << 4;
                break;
            case '\\':
                if ((state == SUFFIX) || (state == NAME))
                {
                    strncpy(name,&tmp_file[pos + 1],SIZE_NAME);
                    tmp_file[pos + 1] = '\0';
                    state = PREFIX;
                }
                break;
            case ':':
                if ((state == SUFFIX) || (state == NAME))
                {
                    strncpy(name,&tmp_file[pos + 1],SIZE_NAME);
                    tmp_file[pos + 1] = '\0';
                    state = DEVICE;
                }
                else
                    if (state == PREFIX)
                    {
                        strncpy(tmp_prefix,&tmp_file[pos + 1],SIZE_PREFIX);
                        tmp_file[pos + 1] = '\0';
                        state = DEVICE;
                    }
                break;
        }
        --pos;                              // go to next character
```

(Listing continued on page 98)

Listing Two *(Listing continued, text begins on page 50.)*

```
} while(pos >= 0);
// now resolve whatever state we ended up in
if ((state == SUFFIX) || (state == NAME))
    strncpy(name,tmp_file,SIZE_NAME);
else
    if (state == PREFIX)
        strncpy(tmp_prefix,tmp_file,SIZE_PREFIX);
    else
        strncpy(device,tmp_file,SIZE_DEVICE);
// validate the device
device[1] = ':';
device[2] = '\0';
if (device[0] == '\0')
    condition |= FLAG_DEVICE;
else
{
    // make the device upper-case for simplicity's sake later on
    device[0] = toupper(device[0]);
    if (device[0] < 'A' || device[0] > 'Z')
        condition |= FLAG_DEVICE << 4;
}
// use the existing change_prefix() method to create the prefix and
// validate it
change_prefix(tmp_prefix);
delete[SIZE_PREFIX + 1] tmp_prefix;
// now validate the name
if (name[0] == '\0')
    condition |= FLAG_NAME;
else
{
    if (name[0] == '.')
    {
        condition |= FLAG_NAME;
        name[0] = '\0';
    }
    else
        if (check_chars(name,FLAG_NAME))
        {
            // as far as we're concerned name HAS to end with a dot.
            i = strlen(name);
            if (name[i - 1] != '.')
            {
                if (i == SIZE_NAME)
                    i = SIZE_NAME - 1;
                name[i++] = '.';
                name[i] = '\0';
            }
        }
    }
    // and suffix
    if (suffix[0] != '\0')
        check_chars(suffix,FLAG_SUFFIX);

}
/*  This constructor creates an empty object suitable for later filling. */
File_Spec::File_Spec(void)
{
    // Set both ends of device, name, and suffix to NUL. Since strncpy()
    // is used later on this guarantees these three are always NUL-terminated.
    device[0] = device[SIZE_DEVICE] = '\0';
    name[0] = name[SIZE_NAME] = '\0';
    suffix[0] = suffix[SIZE_SUFFIX] = '\0';
    prefix = NULL;
    prefix_length = 0;
    request = NULL;
    request_length = 0;
    condition = INCOMPLETE;                 // everthing's incomplete
}
/* The so-called "copy" constructor actually calls a copy() method after
    doing some preliminary initialization.
*/
File_Spec::File_Spec(const File_Spec& original)
{
    prefix = NULL;
    prefix_length = 0;
    request = NULL;
    request_length = 0;
    copy(original);
}
/* The destructor simply releases the memory, if any, assigned to prefix
    and request.
*/
File_Spec::~File_Spec(void)
{
    if (prefix != NULL)
    {
        delete[prefix_length] prefix;
        prefix = NULL;
        prefix_length = 0;
    }
    if (request != NULL)
    {
        delete[request_length] request;
        request = NULL;
        request_length = 0;
    }
```

(Listing continued on page 100)

Listing Two *(Listing continued, text begins on page 50.)*

```
}
/* Tell 'em how we're doing */
unsigned int    File_Spec::status(void)
{
    return(condition);
}
/* This method's purpose is to return a string containing a complete file
    specification string for use by clients.
*/
char    *File_Spec::filespec(void)
{
    int     length;
    // first calculate the length of the file specification
    length = strlen(device);
    length += strlen(prefix);
    length += strlen(name);
    // + 2 to allow for the NUL-terminator and the colon
    length += strlen(suffix) + 2;
    // if request isn't already long enough then de-allocate the current
    // pointer and allocate a new one
    if (request_length < length)
        if (!realloc(&request,&request_length,length))
            return(NULL);
    // build the string
    strcpy(request,device);
    if (prefix != NULL)
        strcat(request,prefix);
    strcat(request,name);
    strcat(request,suffix);
    return(request);
}
/*  get_device(), get_prefix(), get_name(), and get_suffix() are all essen-
    tialy alike. if the request string isn't long enough then it is re-
    allocated, then the attribute that was requested is copied into request.
*/
char    *File_Spec::get_device(void)
{
    errno = 0;
    if (request_length < SIZE_DEVICE + 1)
        if (!realloc(&request,&request_length,SIZE_DEVICE + 1))
            return(NULL);
    strcpy(request,device);
    return(request);
}
/*  Returning the prefix is a bit more complicated than the other get
    routines.
*/
char    *File_Spec::get_prefix(void)
{
    errno = 0;
    if (prefix_length)
    {
        if (request_length < prefix_length)
            if (!realloc(&request,&request_length,prefix_length))
                return(NULL);
        strcpy(request,prefix);
    }
    else
    // even if the prefix is NULL we promised to return something. Here,
    // a string 1 character long consisting of a NUL-terminator
    {
        if (request_length == 0)
            if (realloc(&request,&request_length,1))
                *request = '\0';
    }
    return(request);
}
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
char    *File_Spec::get_name(void)
{
    errno = 0;
    if (request_length < SIZE_NAME + 1)
        if (!realloc(&request,&request_length,SIZE_NAME + 1))
            return(NULL);
    strcpy(request,name);
    return(request);
}
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
char    *File_Spec::get_suffix(void)
{
    errno = 0;
    if (request_length < SIZE_SUFFIX + 1)
        if (!realloc(&request,&request_length,SIZE_SUFFIX + 1))
            return(NULL);
    strcpy(request,suffix);
    return(request);
}
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * /
    as with the get_XXXX() methods above, change_device(), change_prefix(),
    change_name(), and change_suffix() are basically the same. if the
    current condition is "read-only" then return a FALSE. if a NULL string
    is passed (note the default) the current object is truncated and the
    corresponding incomplete flag is set. otherwise SIZE n characters are
    copied and the standard validity checks are made.
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
bool    File_Spec::change_device(const char *string)
{
    if (condition & READ_ONLY)
        return(FALSE);
    if (string == NULL || *string == '\0')
        return(clear_attribute(device,FLAG_DEVICE));
    strncpy(device,string,SIZE_DEVICE);
    device[0] = toupper(device[0]);
    device[1] = ':';
    device[2] = '\0';
    if (device[0] < 'A' || device[0] > 'Z')
    {
        condition |= FLAG_DEVICE << 4;
        return(FALSE);
```

```
    }
    else
        condition &= ~(FLAG_DEVICE << 4);
    condition &= ~FLAG_DEVICE;
    return(TRUE);
}
/*  get_prefix(), like change_prefix(), is somewhat more complicated than the
    other change routines
*/
bool    File_Spec::change_prefix(const char *string)
{
    int     new_length;
    if (condition & READ_ONLY)
        return(FALSE);
    if (string == NULL || *string == '\0')
        return(clear_attribute(prefix,FLAG_PREFIX));
    errno = 0;
    // get the size of the new prefix and if the existing prefix isn't long
    // enough then re-allocate it
    new_length = strlen(string);
    if (new_length > SIZE_PREFIX)
        new_length = SIZE_PREFIX;
    if (prefix_length < new_length + 1)
        if (!realloc(&prefix,&prefix_length,new_length + 1))
            return(FALSE);
    // copy in the new string and validate it.
    strncpy(prefix,string,new_length);
    prefix[new_length] = '\0';
    if (check_chars(prefix,FLAG_PREFIX) == FALSE)
        return(FALSE);
    return(check_prefix());
}
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
bool    File_Spec::change_name(const char *string)
{
    int     i;
    if (condition & READ_ONLY)
        return(FALSE);
    if (string == NULL || *string == '\0')
        return(clear_attribute(name,FLAG_NAME));
    i = 0;
    while (string[i])
    {
        if (string[i] == '.')
        {
            change_suffix(&string[i + 1]);
            if (i == 0)
            {
                name[0] = '\0';
                condition |= FLAG_NAME;
                condition &= ~(FLAG_NAME << 4);
                return(FALSE);
            }
            ++i;
            break;
        }
        if (string[i] == ':' || string[i] == '\\')
        {
            condition |= FLAG_NAME << 4;
            return(FALSE);
        }
        if (++i == SIZE_NAME)
            break;
    }
    strncpy(name,string,i);
    name[i] = '\0';
    if (check_chars(name,FLAG_NAME) == FALSE)
        return(FALSE);
    i = strlen(name);
    // as far as we're concerned name HAS to end with a dot.
    if (name[i - 1] != '.')
    {
        if (i == SIZE_NAME)
            i = SIZE_NAME - 1;
        name[i++] = '.';
        name[i] = '\0';
    }
    condition &= ~FLAG_NAME;
    return(TRUE);
}
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
bool    File_Spec::change_suffix(const char *string)
{
    if (condition & READ_ONLY)
        return(FALSE);
    if (string == NULL)
    {
        clear_attribute(suffix,FLAG_SUFFIX);
        condition &= ~FLAG_SUFFIX;      // unset the incomplete suffix flag
    }
    if (*string == '.')
    {
        ++string;
        strncpy(suffix,string,SIZE_SUFFIX);
    }
    else
        strncpy(suffix,string,SIZE_SUFFIX);
    if (check_chars(suffix,FLAG_SUFFIX) == FALSE)
        return(FALSE);
    condition &= ~FLAG_SUFFIX;
    return(TRUE);
}
/*  This method determines whether or not the prefix is complete. */
bool    File_Spec::check_prefix(void)
{
    int     i;
```

(Listing continued on page 102)

Listing Two *(Listing continued, text begins on page 50.)*

```
    // if the 1st character isn't a '\' then the prefix is relative to the
    // current working directory.
    if (prefix[0] != '\\')
    {
        condition |= FLAG_PREFIX;
        return(FALSE);
    }
    i = 0;
    // this loop checks for the presence of a dot followed by another dot
    // or a dot followed by a backslash. either one indicates the prefix
    // is relative the the current working directory.
    while (prefix[i + 1])
    {
        if ((prefix[i] == '.') &&
          (prefix[i + 1] == '\\' || prefix[i + 1] == '.'))
            {
                condition |= FLAG_PREFIX;
                return(FALSE);
            }
        if (++i == SIZE_PREFIX - 1)
            break;
    }
    // a prefix HAS to end with a '\'
    if (prefix[i] != '\\')
    {
        prefix[i++] = '\\';
        prefix[i] = '\0';
    }
    condition &= ~FLAG_PREFIX;
    return(TRUE);
}
/* * * *
```

Listing Two *(Listing continued, text begins on page 50.)*

```
    }
    // DOS doesn't append a '\' either so we will
    cwd_elem = strlen(cwd);
    if (cwd[1] != '\0')
    {
        cwd[cwd_elem] = '\\';
        cwd[cwd_elem + 1] = '\0';
    }
    // if there was no prefix, there is now. assign it and return
    if (prefix == NULL)
    {
        prefix_length = SIZE_PREFIX + 1;
        prefix = cwd;
        return(TRUE);
    }
    prefix_elem = 0;
    state = 0;
    do
    {
        switch(state)
        {
            case 0:
                if (prefix[prefix_elem] == '.')
                    // a dot means check for another dot or '\'. goto state 1
                    state = 1;
                else
                    // DOS would give you a fit over this. since we're here,
                    // check_prefix() found a relative component in the path.
                    // however, the initial '\' means "start at the root".
                    // so we go to the root by setting cwd_elem to zero and
                    // start looking for a dot.
                    if (prefix[prefix_elem] == '\\')
                    {
                        state = 1;
                        cwd_elem = 0;
                    }
                    else
                    {
                      // our current character IS a character so get ready to
                      // append it to cwd by going to state 3 and backing up
                      // so we don't lose it.
                      state = 3;
                      --prefix_elem;
                    }
                break;
            case 1:                         // we have seen a dot (or a '\')
                if (prefix[prefix_elem] == '.')
                    // another dot means go up a directory. enter state 2.
                    state = 2;
                else
                    // a '\' means stay here. get ready to append to the
                    // cwd and enter state 3.
                    if(prefix[prefix_elem] == '\\')
                        state = 3;
                    else
                        // a character here means we just saw something like
                        // .s - remain in the current directory and pass the
                        // buck back to state 0.
                        state = 0;
                break;
            case 2:                         // two (or more) dots in a row
                if (prefix[prefix_elem] == '\\')
                {
                    if (cwd_elem > 0)
                        do
                        {
                            --cwd_elem;
                        } while (cwd[cwd_elem] != '\\');
                }
                else
                    // more than two dots in a row. maintain current state
                    if (prefix[prefix_elem] == '.')
                        break;
                    else
                        // this means we're seeing a "..s" type situation.
                        // treat it as a "..\s" and back up one so we don't
                        // lose the prefix character.
                        --prefix_elem;
                state = 3;
                break;
            case 3:                         // append the prefix to the cwd
                if (prefix[prefix_elem] == '.')
                    // whoops, another dot. go back to state 1
                    state = 1;
                else
                    // more than one '\' in a row. don't change state.
                    if (prefix[prefix_elem] == '\\')
                        break;
                    else
                    {
                        // the order of element increments is a bit peculiar
                        // but remember we've been moving in opposite
                        // directions
                        do
                        {
                            ++cwd_elem;
                            cwd[cwd_elem] = prefix[prefix_elem];
                            ++prefix_elem;
                        } while (prefix[prefix_elem] != '\\');
                        ++cwd_elem;
                        cwd[cwd_elem] = prefix[prefix_elem];
                    }
                break;
        };
        ++prefix_elem;
    } while (prefix[prefix_elem]);
    cwd[++cwd_elem] = '\0';
    // it worked! reset the prefix pointer and get out
    delete[prefix_length] prefix;
    prefix = cwd;
    prefix_length = SIZE_PREFIX + 1;
    return(TRUE);
}
/* This routine checks for valid DOS file name characters. */
bool    File_Spec::check_chars(char *string, unsigned int attrib_flag)
{
    while (*string)
    {
        if ((*string < '!') ||
            (*string == '"') ||
            (*string > ')' && *string < '-') ||
            (*string == '/') ||
            (*string > '9' && *string < '@') ||
            (*string == '[') ||
            (*string > '\\' && *string < '^') ||
            (*string == ':'))
        {
            condition |= attrib_flag << 4;
            return(FALSE);
        }
        ++string;
    }
    condition &= ~(attrib_flag << 4);
    return(TRUE);
}
/* Since two methods (the 2nd constructor and the "=" operator) need to
    make copies of other File_Spec instances, this private method is provided
    to avoid duplicating the code. this method is also the main reason for
    making File_Spec a friend of itself.
*/
void    File_Spec::copy(const File_Spec& original)
{
    errno = 0;
    strcpy(device,original.device);
    if (prefix_length < original.prefix_length)
        if (!realloc(&prefix,&prefix_length,original.prefix_length))
            return;
    strcpy(prefix,original.prefix);
    strcpy(name,original.name);
    strcpy(suffix,original.suffix);
    condition = original.condition;
}
/*  clear_attribute sets the first element of the attribute it is passed to
    a NUL and then resets the condition flags.
*/
bool    File_Spec::clear_attribute(char *attribute, unsigned int attrib_flag)
{
    if (attribute != NULL)
        *attribute = '\0';
    condition |= attrib_flag;
    condition &= ~(attrib_flag << 4);
    return(TRUE);
}
/* realloc() is used by the request and prefix attributes when they're changed
    to see if they require re-allocating and to handle the re-allocation
    if needed.
*/
bool    File_Spec::realloc(char **pointer, int *length, int new_length)
{
    if (*length)
        delete[*length] *pointer;
    if ((*pointer = new char[new_length]) == NULL)
    {
        *length = 0;
        *pointer = NULL;
        return(FALSE);
    }
    *length = new_length;
    return(TRUE);
}
```

End Listing Two

Listing Three

```
/* FILE.HPP Written by Kevin D. Weeks Released to the Public Domain  */

#ifndef FILE_HPP                            // prevent multiple #includes
#define FILE_HPP

#include "filespec.hpp"

// file access mode definitions
#define F_RDONLY    0x0000
#define F_WRONLY    0x0001
#define F_RDWR      0x0002
#define F_COMPAT    0x0000
#define F_DENYALL   0x0010
#define F_DENYWR    0x0020
#define F_DENYRD    0x0030
#define F_DENYNO    0x0040

// flag to determine whether reads and writes have a side-effect on the
// file pointer
#define F_ADVANCE   0x0100

class   File: public File_Spec
{
```

(Listing continued on page 108)

Listing Three *(Text begins on page 50.)*

```
    // class attributes
    int         handle;                 // DOS file handle
    int         open_flags;             // flags used to open or create
    long        file_pos;               // DOS file position
    long        filelength;             // length of file
  // public class methods
  public:
    // constructors for file objects
            File(void);
            File(const File_Spec& original, bool open_flag = FALSE);
            File(const char *name, bool open_flag = FALSE);
    // destroy the object
            ~File(void);
    bool    exists(void);               // see if the file exists
    bool    create(int mode_flags = 2, bool exclusive = TRUE);
    bool    open(int mode_flags = 2);
    bool    close(void);
    unsigned int    read(void *buffer, unsigned int size);
    // guarantees to write size bytes or fail
    bool    write(const void *buffer, unsigned int size);
    bool    truncate(void);             // truncate file at current position
    // guarantees to position file pointer within file or fail
    bool    set_position(long new_file_pos);
    long    get_position(void);
    long    size(void);
    bool    rename(const char *newname);
    bool    erase(void);
    File    *copy(const char *newfile, bool overwrite = FALSE);
};
#endif
```

End Listing Three

Listing Four

```
/* FILE.CPP Written by Kevin D. Weeks Released to the Public Domain  */

#include <errno.h>
#include <io.h>
#include <sys\stat.h>
#include <dos.h>
#include "file.hpp"

// this declaration instructs the compiler to NOT perform name-mangling
// on these functions.
extern "C"
{
    extern char     ll_get_drive(void);
    extern int      ll_get_cwd(int, char *);
    extern unsigned int ll_write(int, unsigned int, const void *);
}

extern volatile int errno;

/*  An empty file object seems silly but here it is anyway */
File::File(void)
{
    handle = -1;
    file_pos = 0L;
    open_flags = 0;
    filelength = 0L;
}

/*  This is the File version of the "copy" constructor. it is posible to
    open the file when the object is instantiated by passing TRUE as a
    second parameter.
*/
File::File(const File_Spec& original, bool open_file):(original)
{
    handle = -1;
    file_pos = 0L;
    open_flags = 0;
    if ((filelength = filesize(filespec())) == -1L)
        filelength = 0L;
    if (open_file == TRUE)
        open();
}

/*  This constructor is the same as the one above. The differences in pass
    parameters are handled by their respective ancestors.
*/
File::File(const char *name, bool open_file):(name)
{
    handle = -1;
    file_pos = 0L;
    open_flags = 0;
    if ((filelength = filesize(filespec())) == -1L)
        filelength = 0L;
    if (open_file == TRUE)
        open();
}

/*  File destructor */
File::~File(void)
{
    if (handle > 0)
        close();
}

/*  Does the file exist? */
bool    File::exists(void)
{
    if (!complete())                    // check for a completed file spec
        return(FALSE);                  // and either fail if not
    if (findfirst(filespec(),0) == NULL)  // or else check for directory entry
        return(FALSE);
    return(TRUE);
}

/* Create the file. */
bool    File::create(int mode_flags, bool exclusive)
{
    int     tmp_handle;
    if (!complete())                    // is the file spec complete?
        return(FALSE);
    if (handle > -1)                    // if the file is open
    {
        if (exclusive)
            return(FALSE);
        else
            close();
    }
    else
        if (exists())                   // if the file exists
            if (exclusive)              // if this flag is TRUE
            {                           // return an error
                errno = EEXIST;
                return(FALSE);
            }
    // create the the file and then close it to re-open with the appropriate
    // mode flags set
    if ((tmp_handle = creat(filespec(),S_IWRITE | S_IREAD)) == ERR)
        return(FALSE);
    ::close(tmp_handle);                // the :: means use the library close
    if (open(mode_flags) == FALSE)      // no :: - use the File method
        return(FALSE);
    set_position(0L);                   // position at the beginning
    filelength = 0L;
    read_only(TRUE);                    // tell File_Spec that it can't
                                        // be changed
    return(TRUE);
}

/* Open the file. */
bool    File::open(int mode_flags)
{
    if (handle > -1)                    // if the file is already open
        return(TRUE);                   //      don't re-open it
    if (!complete())                    // check for a complete file spec
        return(FALSE);
    // use the standard library to actually open it ( ::open(...) )
    if ((handle = ::open(filespec(),mode_flags)) == ERR)
        return(FALSE);
    open_flags = mode_flags;            // keep the mode flags
    read_only(TRUE);                    // tell File_Spec not to change a
                                        //      thing
    return(TRUE);
}

/* Close the file. */
bool    File::close(void)
{
    if (handle > -1)                    // if the file's open
        if (::close(handle) == ERR)     //     close it
            return(FALSE);
    handle = -1;                        // and re-initialize everything
    file_pos = 0L;
    open_flags = 0;
    read_only(FALSE);                   // File_Spec can change again
}

/*  Read the file. NOTE: bytes actually read may be less than requested.  */
unsigned int    File::read(void *buffer, unsigned int num_bytes)
{
    int     bytes_read;
    if (handle < 0)                     // make sure the file's open
    {
        errno = EBADF;
        return(FALSE);
    }
    // first set the file position. if auto-advance is on set_position()
    // will just return. otherwise it will move the file pointer to where
    // it should be. then use the standard read to read the file
    if (set_position(file_pos) != FALSE)
        if ((bytes_read = ::read(handle,buffer,num_bytes)) == ERR)
            return(FALSE);
    // if auto-advance is on we still need to keep ourselves current
    if (open_flags & F_ADVANCE)
        file_pos += (long)bytes_read;
    // return the number of bytes actually read
    return(bytes_read);
}

/*  Write to the file. In this case failure to write the number of bytes
    specified IS considered a failure.
*/
bool    File::write(const void *buffer, unsigned int num_bytes)
{
    if (handle < 0)                     // is the file open?
    {
        errno = EBADF;
        return(FALSE);
    }
    if (num_bytes == 0)                 // if zero bytes are to be written
        return(TRUE);                   // return WITHOUT truncating the file
    // make sure the file pointer is positioned right and then call our
    // low level write routine to write it. there's no reason to clutter up
    // the program with the library write()
```

(Listing continued on page 110)

Listing Four *(Listing continued, text begins on page 50.)*

```
    if (set_position(file_pos) != FALSE)
        if (ll_write(handle,num_bytes,buffer) < num_bytes)
        {
            // at this point we failed to write as many bytes as desired. to
            // eliminate side effects we truncate
            truncate();
            return(FALSE);
        }
    // if we wrote at the end of the file, increase its length
    if (file_pos == filelength)
        filelength += (unsigned long)num_bytes;
    if (open_flags & F_ADVANCE)             // check for auto-advance
        file_pos += (long)num_bytes;
    return(TRUE);
}

/*  Truncate chops a file off at the current file_position. */
bool    File::truncate(void)
{
    if (handle < 0)                         // don't bother if we're not open
    {
        errno = EBADF;
        return(FALSE);
    }
    // re-set the file pointer and write zero bytes
    if (set_position(file_pos) != ERR)
        if (!ll_write(handle,0,NULL))
            return(FALSE);
    filelength = file_pos;                  // re-set the length
    return(TRUE);
}

/*  Position the DOS file pointer */
bool    File::set_position(long new_file_pos)
{
    if (handle < 0)                         // guess!
    {
        errno = EBADF;
        return(FALSE);
    }
    // first make sure we're not attempting to set before the beginning or
    // after the end of the file.
    if (new_file_pos > filelength || new_file_pos < 0L)
        return(FALSE);
    // position it
    if (lseek(handle,new_file_pos,SEEK_SET) == -1L)
        return(FALSE);
    file_pos = new_file_pos;
    return(TRUE);
}

/*  Get the current file position */
long    File::get_position(void)
{
    return(file_pos);
}

/*  Get the file size */
long    File::size(void)
{
    long    length;
    if (handle > -1)
        return(filelength);
    else
    {
        if (open() == FALSE)
            return(0L);
        length = filelength;
        close();
        return(length);
    }
}

/*  If we attempt to rename an open file it is first closed and then
    reopened after the rename.
 */
bool    File::rename(const char *newname)
{
    bool    reopen = FALSE;
    int     tmp_flags;
    int     i;
    if (handle > -1)                        // close the file if it's open
    {
        tmp_flags = open_flags;
        close();
        reopen = TRUE;
    }
    else
        if (exists() == FALSE)              // make sure the file exists
            return(FALSE);
    // create a new file spec just like this one (note that it's also
    // instantiated at this point)
    File_Spec newspec = *this;
    newspec.change_name(newname);           // and then change the name
    if (::rename(filespec(),newspec.filespec()) != 0)
    {
        if (reopen)
            // pass the existing open_flags in case the default wasn't used
            // when the file was originally opened.
            open(tmp_flags);
        return(FALSE);
    }
    change_name(newname);                   // now update this file name
    if (reopen)
        return(open(tmp_flags));
    return(TRUE);
}

/*  Erase the file. if it's open, close it first. */
bool    File::erase(void)
{
    if (handle > -1)
        close();
    if (unlink(filespec()) == ERR)
        return(FALSE);
    return(TRUE);
}

/*  This might also be a good opportunity for operator overloading.  */
File    *File::copy(const char *newname, bool overwrite)
{
    File            *newfile;               // file to copy to
    char            *buffer;                // I/O buffer
    unsigned int    buf_size;               // size of I/O buffer
    unsigned int    num_bytes;              // number of bytes transfered
    long            tmp_file_pos;           // temporary file position holder
    bool            re_close = FALSE;       // flag indicating if source file
                                            // should be closed following the
                                            // copy (to avoid side effects)
    int             tmp_old_flags;          // the original source open flags

    errno = 0;
    // first create the new object instance
    if ((newfile = new File(newname)) == NULL)
    {
        errno = ENOMEM;
        return(NULL);
    }
    // then create the new file (invert overwrite for create)
    if (!(newfile->create(F_ADVANCE | F_RDWR,(bool)!overwrite)))
    {
        delete newfile;
        return(NULL);
    }
    // attempt to allocate a buffer. loop until successful or fail at 1 char
    buf_size = 32768;
    while ((buffer = new char[buf_size]) == NULL)
    {
        buf_size /= 2;
        if (buf_size == 1)
        {
            errno = ENOMEM;
            delete newfile;
            return(NULL);
        }
    }
    if (handle < 0)                         // if the source file isn't open
    {
        if (!open())                        // open it
        {
            newfile->close();               // if we can't open the source
            newfile->erase();               // file we need to clean up
            delete newfile;
            delete[buf_size] buffer;
            return(NULL);
        }
        re_close = TRUE;                    // copy() opened it so copy()
    }                                       // should close it
    tmp_old_flags = open_flags;             // keep the original open flags
    open_flags |= F_ADVANCE;                // and turn auto-advance on
    tmp_file_pos = file_pos;                // keep the original file pointer
    set_position(0L);                       // go to the beginning of the file
    // loop until the entire file has been copied
    while (num_bytes = read(buffer,buf_size))
    {
        if (newfile->write(buffer,num_bytes) == FALSE)
        {                                   // if a write error occurs we
            newfile->close();               // need to clean up the mess
            newfile->erase();               // and return an error
            delete newfile;
            delete[buf_size] buffer;
            open_flags = tmp_old_flags;
            set_position(tmp_file_pos);
            if (re_close)
                close();
            return(NULL);
        }
    }
    // clean up and return the new file
    newfile->close();
    delete[buf_size] buffer;
    open_flags = tmp_old_flags;
    set_position(tmp_file_pos);
    if (re_close)
        close();
    return(newfile);
}
```

End Listing Four

Listing Five

```
;*****************************************************************************
;   LOWIO.ASM Written by Kevin D. Weeks Released to the Public Domain
;

include MACROS.ASM                          ; macro file provided by Zortech

; import errno
begdata
    extrn  _errno:word
enddata

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;   bool    ll_write(int file_handle, unsigned int num_bytes, void *buffer);
;   ll_write simply makes a call to DOS for a write. it varies in two ways
;   from the standard C write().
;   1. the order of pass parameters (to simplify dealing with 80x86 segments)
;   2. it WILL truncate a file
;
begcode ll_write
    c_public ll_write
func ll_write
    push    bp
    mov     bp,sp
    push    bx
    push    cx
    push    dx
    mov     bx,P[bp]                        ; get file handle from stack
    mov     cx,P[bp + 2]                    ; get number of bytes to write
    mov     dx,P[bp + 4]                    ; get offset of buffer
if LPTR                                     ; if large memory model
    mov     ds,P[bp + 6]                    ; get segment of buffer
endif
    mov     ax,4000h                        ; dos write file function
    int     21h                             ; call dos
    jc      write_err                       ; carry flag indicates error
    jmp     write_ret
write_err:
    mov     _errno,ax                       ; set errno to error

write_ret:
    pop     ds
    pop     dx
    pop     cx
    pop     bx
    mov     sp,bp
    pop     bp
    ret
c_endp  ll_write
endcode ll_write

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;   int     ll_get_drive(void);
;   ll_get_drive simply returns the current looged disk drive. there is no
;   error return.
;
begcode ll_get_drive
    c_public ll_get_drive
func ll_get_drive
    push    bp
    mov     bp,sp
    mov     ax,1900h                        ; dos get current drive function
    int     21h                             ; call dos
    xor     ah,ah                           ; clear high byte
    mov     sp,bp
    pop     bp
    ret
c_endp  ll_get_drive
endcode ll_get_drive

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;   bool    ll_get_cwd(int drive);
;   ll_get_cwd gets the current working directory for the specified drive.
;   0 means the current drive, 1 means drive A, 2 means drive B, etc. it
;   returns a 0 if an error occurs and errno is set.
;
begcode ll_get_cwd
    c_public ll_get_cwd
func ll_get_cwd
    push    bp
    mov     bp,sp
    push    dx
    push    si
    push    ds
    mov     dx,P[bp]                        ; get drive
    mov     si,P[bp + 2]                    ; get offset of buffer
if LPTR                                     ; if large memory model
    mov     ds,P[bp + 4]                    ; get segment of buffer
endif
    mov     ax,4700h                        ; dos get current cwd function
    int     21h                             ; call DOS
    jc      get_cwd_err                     ; carry flag indicates error
    xor     ax,ax
    jmp     get_cwd_ret
get_cwd_err:
    mov     _errno,ax                       ; set errno to error
    mov     ax,0ffffh                       ; & set ax to -1
get_cwd_ret:
```

(Listing continued on page 112)

Listing Five *(Listing continued, text begins on page 50.)*

```
    pop     ds
    pop     si
    pop     dx
    mov     sp,bp
    pop     bp
    ret
c_endp  ll_get_cwd
endcode ll_get_cwd

END
```

End Listing Five

Listing Six

```
/* FILETEST.CPP Written by Kevin D. Weeks Released to the Public Domain  */

#include <stdio.h>
#include <errno.h>
#include <string.h>
#include "file.hpp"

// File_Spec test cases
char    *device_test[] =
{
    "",                                     // no device
    "a:",                                   // complete device
    ">:",                                   // invalid file char
    "ab:",                                  // device too long
    "a::",                                  // invalid char (double colon)
    NULL
};

char    *prefix_test[] =
{
  "",                                       // no prefix
  "\\sub\\",                                // complete prefix
  "\\>sub\\",                               // complete with invalid file char
  // prefix too long

"\\0123456789\\0123456789\\0123456789\\0123456789\\0123456789\\0123456789\\",
  "sub.dir\\",                              // incomplete prefix (no backslash)
  "\\\\sub\\",                              // double back slash
  "\\sub.dir\\",                            // complete prefix with extension
  "sub1\\sub2\\",                           // bi-level incomplete prefix
  "\\sub1\\sub2\\",                         // bi-level conplete prefix
  "..\\sub",                                // relative, incomplete prefix
  "..\\sub\\",                              //     "   ,      "       "
  "..\\",                                   //     "   ,      "       "
  ".\\sub\\",                               //     "   ,      "       "
  "\\..\\sub\\",                            // relative but starts at root
  "..\\>sub\\",                             // relative with invalid file char
  "\\",                                     // complete prefix
  NULL
};

char    *name_test[] =
{
    "",                                     // no name
    "file.",                                // complete name
    "file>.",                               // complete with invalid file char
    "filetest1.",                           // name too long
    "file..",                               // invalid char (double dot)
    "file",                                 // incomplete, no dot
    NULL
};

char    *suffix_test[] =
{
    "",                                     // no suffix
    ".tst",                                 // complete suffix
    ".ts>",                                 // complete with invalid file
char
    ".tst1",                                // suffix too long
    NULL
};

char    **test[] =
{
    device_test,
    prefix_test,
    name_test,
    suffix_test
};

char    *test_type[] =
{
    "DEVICE ",
    "PREFIX ",
    "NAME ",
    "SUFFIX "
};

extern volatile int  errno;
void    check_file_spec(void);
void    check_file(void);
void    make_filespec(char *test_case);
void    print_condition(File_Spec& file,char *test_name,char *test_case);

int     main(void)
{
    check_file_spec();
    check_file();
}

void    check_file_spec()
{
    int     i, j, k;
    char    test_case[81];
    char    title[81];

    printf("\nTESTING File_Spec...\n\n");
    File_Spec   file1;                      // check void constructor
    print_condition(file1,"void constructor"," ");

    for (i = 0; i < 4; i++)
    {
        printf("CHECKING %s\n",test_type[i]);
        j = 0;
        while (test[i][j] != NULL)
        {
            make_filespec(test[i][j]);
            ++j;
        }
    }

    // check first four complete combinations
    printf("\nCHECKING COMPLETE FILE SPECS\n");
    for (i = 0; i < 4; i++)
    {
        strcpy(test_case,test[0][i]);
        strcat(test_case,test[1][i]);
        strcat(test_case,test[2][i]);
        strcat(test_case,&test[3][i][1]);
        make_filespec(test_case);
    }

    for (i = 0; i < 4; i++)
    {
        printf("CHECKING change_%s\n",test_type[i]);
        j = 0;
        while (test[i][j] != NULL)
        {
            switch (i)
            {
                case 0:
                    if (file1.change_device(test[i][j]) == FALSE)
                        printf("Error changing device");
                    break;
                case 1:
                    if (file1.change_prefix(test[i][j]) == FALSE)
                        printf("Error changing prefix");
                    break;
                case 2:
                    if (file1.change_name(test[i][j]) == FALSE)
                        printf("Error changing name");
                    break;
                case 3:
                    if (file1.change_suffix(test[i][j]) == FALSE)
                        printf("Error changing suffix");
                    break;
            };
            print_condition(file1," ",test[i][j]);
            ++j;
        }
    }
    printf("\nCOMPLETION TEST\n");
    file1.change_device();                  // erase current device
    file1.change_prefix();                  // & prefix
    print_condition(file1,"BEFORE"," ");
    file1.complete();
    print_condition(file1,"AFTER"," ");

    File_Spec file2 = file1;
    print_condition(file2,"\nTEST '=' OPERATOR\n","file2 = file1");

}

void    make_filespec(char *test_case)
{
    File_Spec file2(test_case);
    print_condition(file2,"char constructor",test_case);
    File_Spec file3(file2);
    print_condition(file3,"copy constructor",test_case);
}

void    print_condition(File_Spec& file, char *test_name, char *test_case)
{
```

(Listing continued on page 115)

Listing Six *(Listing continued, text begins on page 50.)*

```
    unsigned int    status;
    char            *completion;
    char            *character;
    static char     incomplete[] = {"INCOMPLETE"};
    static char     complete[] = {"  complete"};
    static char     invalid[] = {"INVALID CHAR"};
    static char     valid[] = {"  chars ok  "};

    printf("%s\t%s\n",test_name,test_case);
    status = file.status();
    printf("file condition: %x\n",status);

    completion = (status & FLAG_DEVICE) ? incomplete : complete;
    character = (status & (FLAG_DEVICE << 4)) ? invalid : valid;
    printf("\tdevice: %-9s\t%s\t%s\n",file.get_device(),completion,character);

    completion = (status & FLAG_PREFIX) ? incomplete : complete;
    character = (status & (FLAG_PREFIX << 4)) ? invalid : valid;
    printf("\tprefix: %-9s\t%s\t%s\n",file.get_prefix(),completion,character);

    completion = (status & FLAG_NAME) ? incomplete : complete;
    character = (status & (FLAG_NAME << 4)) ? invalid : valid;
    printf("\t  name: %-9s\t%s\t%s\n",file.get_name(),completion,character);

    character = (status & (FLAG_SUFFIX << 4)) ? invalid : valid;
    printf("\tsuffix: %-9s\t%s\t%s\n",file.get_suffix()," ",character);

    printf("\tfilespec: %s\n\n",file.filespec());
}

void    check_file(void)
{
    char    buffer[81];
    int     i;

    printf("\n\n\nTESTING File...\n\n\n");

    /* we won't try to perform any constructor tests since most of the
       attributes are in-accessable and therefore best checked using
       either a source-level debugger or printf statements. */
    File file1("file.tst");

    printf("Creating %s\n",file1.filespec());
    if (file1.create() == FALSE)
    {
        printf("%s already exists. Re-creating it.\n",file1.filespec());
        if (file1.create(F_RDWR,FALSE) == FALSE)
        {
            printf("Error re-creating %s\n",file1.filespec());
            perror("");
            return;
        }
    }
    printf("File %s successfully created and opened\n",file1.filespec());

    strcpy(buffer,"this is a test file");
    i = strlen(buffer);
    if (file1.write(buffer,i) == FALSE)
    {
        perror("Error writing");
        printf("Closing file\n");
        return;
    }
    printf("\"%s\" written to file\n",buffer);
    printf("Current file position is: %ld\n",file1.get_position());
    printf("Current file length is: %ld\n",file1.size());
    if (file1.read(buffer,i) == FALSE)
    {
        perror("Error reading");
        printf("Closing file\n");
        return;
    }
    printf("\"%s\" read from file\n",buffer);
    if (file1.rename("test.fil") == FALSE)
    {
        perror("Error renaming");
        printf("Closing file\n");
        return;
    }
    printf("File renamed to %s\n",file1.filespec());

    File *file2 = file1.copy("test2.fil");
    if (errno)
    {
        perror("test2.fil");
        printf("Overwriting it.\n");
        file2 = file1.copy("test2.fil",TRUE);
  }
  if (file2->exists())
    printf("%s successfully copied to %s\n",file1.filespec(),file2-
>filespec());
  else
  {
        printf("Copy failed.\n");
        return;
    }

    delete file2;

}
```

End Listings

Listing One *(Text begins on page 56.)*

```
/* points.c    Copyright (c) 1989 by Norton T. Allen */

#include "points.h"

int nbits(int i) {
  int nb;

  for (nb = 0; i != 0; nb++) i >>= 1;
  return(nb);
}

static int focus_x, focus_y, focus_width, focus_height;

/* zeros in map correspond to x bits, 1's to y bits */
static long int z, map, zlim;
static int tbits, xor_y;

void init_points(int left, int top, int width, int height) {
  int nbits_w, nbits_h;
  long int mask;

  focus_x = left;
  focus_y = top;
  focus_width = width;
  focus_height = height;
  nbits_w = nbits(focus_width);
  nbits_h = nbits(focus_height);
  tbits = nbits_w + nbits_h;
  xor_y = nbits_w < nbits_h;
  z = 0L;
  zlim = 1L << tbits;
  map = 0L;
  for (mask = 1L; nbits_w || nbits_h; ) {
    if (nbits_w > nbits_h || (nbits_w == nbits_h && xor_y)) {
      nbits_w--;
      mask <<= 1;
    }
    if (nbits_h > 0 &&
        (nbits_h > nbits_w || (nbits_w == nbits_h && !xor_y))) {
      nbits_h--;
      map |= mask;
      mask <<= 1;
    }
  }
}

int next_point(int *dx, int *dy) {
  int x, y, n;
  long int m, rz;

  for (;;) {
    if (z == zlim) return(1);
    m = map; rz = z; n = tbits;
    x = y = 0;
    while (n-- > 0) {
      if (m & 1) { /* this is a ybit */
        y <<= 1;
        if (rz & 1) y++;
      } else {
        x <<= 1;
        if (rz & 1) x++;
      }
      m >>= 1; rz >>= 1;
    }
    z++;
    if (xor_y) y = x ^ y;
    else x = x ^ y;
    if (x >= focus_width) continue;
    if (y >= focus_height) continue;
    break;
  }
  *dx = x + focus_x;
  *dy = y + focus_y;
  return(0);
}
```

End Listing One

Listing Two

```
/* mandel.c will handle the actual Mandelbrot set calculations.
   Copyright (c) 1989 by Norton T. Allen */

#include "grafix.h"

/* This is the aspect ratio calculated on my screen: */
#define ASPECT 0.739

static double xscale, yscale, xo, yo;

/* set_range() is similar to setcoords() except that it guarantees
   equal x and y scales, taking the aspect ratio into account.
   The axis with the smaller scale is adjust so the specified
   range will be centered on the screen.  Another reason for
   not using setcoords() is that I need the inverse functions
   mapping device coordinates to virtual coordinates.
*/
void set_range(double xmin, double ymin, double xmax, double ymax) {
  double delta;

  yscale = (ymax-ymin)/vp_height();
  xscale = (xmax-xmin)/vp_width();
  if (yscale*ASPECT > xscale) {
    xscale = yscale*ASPECT;
    delta = (xmin + xscale * vp_width() - xmax)/2.;
    xmin -= delta;
    xmax += delta;
  } else {
    yscale = xscale/ASPECT;
    delta = (ymin + yscale * vp_height() - ymax)/2.;
    ymin -= delta;
    ymax += delta;
  }
  xo = xmin;
  yo = ymin;
} /* ------------------------------------------------------------------ */

double vx (int dx)              /* convert device x to virtual x */
{
  return ((double) dx * xscale + xo);
} /* ------------------------------------------------------------------ */

double vy (int dy)              /* convert device y to virtual y */
{
  return ((double) dy * yscale + yo);
} /* ------------------------------------------------------------------ */

/* Membership in the Mandelbrot set is determined by an
   iterative function.  Given the complex starting
   coordinate C the function is:
                Z(0) = 0.
                Z(n+1) = Z(n)^2 + C
   A point is deemed to be in the set if NLOOP iterations fails
   to produce a point with absolute value greater than LIMIT.
   Points within the set are colored black.  Points outside the
   set are colored based on how many iterations passed before
   exceeding LIMIT.  Miriad other schemes are possible.  Other
   values for NLOOP and LIMIT are probably desirable, depending
   on how deep you go.
*/
#define NLOOP 100
#define LIMIT 10000.
#define NCOLORS 15

int mandel_color(double cx, double cy) {
  int k;
  double zx, zy, zx2, zy2;

  zx = zy = 0.;
  for (k = 0; k < NLOOP; k++) {
    zx2 = zx*zx;
    zy2 = zy*zy;
    if (zx2+zy2 > LIMIT) break;
    zy = 2*zx*zy + cy;
    zx = zx2 - zy2 + cx;
  }
  if (k < NLOOP) return((k % NCOLORS)+1);
  return(0);
}
```

End Listing Two

Listing Three

```
/* generate.c    Copyright (c) 1989 by Norton T. Allen */

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include "points.h"
#include "grafix.h"

/* This structure defines a rectangular box which we can move around the
   screen using the cursor keys.  fbox.on is TRUE if the box is
   currently displayed.  fbox.mode takes the values:
                0       Not active
                1       Cursor keys move the whole box
                2       Cursor keys change box's size
*/
struct bx {
  int x, y, dx, dy, on, mode;
} fbox = {300, 160, 50, 40, 0, 0};

void flip_box(void) {
  set_write_mode(WM_XOR);
  set_color1(15);
  draw_rect(fbox.x, fbox.y, fbox.dx, fbox.dy);
  set_write_mode(WM_REPLACE);
  fbox.on = !fbox.on;
}

void help(void) {
  printf("'Esc' to exit\n");
  printf("?     for this message\n");
  printf("m     to move the focus box.(use cursor keys)\n");
  printf("s     to size the focus box.(use cursor keys)\n");
  printf("f     to focus on the focus area\n");
  printf("z     to zoom in on the focus area\n");
  getch();
}
static int dm = 1;      /* How many pixels to move with each cursor step */

/* check_box makes sure the new box meets the following requirements:
     1. It isn't off the screen.
     2. It's not smaller then 2 pixels in either dimension
   Check box turns off the old box, but leaves the new box off also;
   menu() will turn it on when there's no more keyboard input.
*/
void check_box(struct bx *nb) {
  if (nb->x < 0 || nb->y < 0 ||
      nb->x + nb->dx > vp_width() ||
```

Listing Three *(Listing continued, text begins on page 56.)*

```
      nb->y + nb->dy > vp_height() ||
      nb->dx < 2 || nb->dy < 2 ||
      (nb->x == fbox.x && nb->y == fbox.y &&
       nb->dx == fbox.dx && nb->dy == fbox.dy))
    return;
  if (fbox.on) flip_box();
  fbox = *nb;
  fbox.on = 0;
}

/* These are scan codes for cursor keys: */
#define EX_UP 72
#define EX_DOWN 80
#define EX_RIGHT 77
#define EX_LEFT 75

/* Menu supports the following keys:
    ESC         Exit
    M           suspend calculations and put box on the screen in
                mode 1 for 'moving'.
    S           As with M, but mode 2 for 'sizing'.
    C           Remove box and continue calculations as before
    F           Focus on boxed region.
    Z           Zoom in on boxed region.
    Cursor keys Move or size box
    0-9         Change step size for cursor keys
*/
void menu(void) {
  int c;
  struct bx new_box;
  double xmin, ymin, xmax, ymax;

  for (;;) {
    while (kbhit()) {
      c = getch();
      switch (c) {
        case '\033':
          exit(0);
        case 'c':
        case 'C':
          fbox.mode = 0;
          break;
        case 'm':
        case 'M':
          fbox.mode = 1;
          break;
        case 's':
        case 'S':
          fbox.mode = 2;
          break;
        case 'f': /* Focus on boxed region */
        case 'F':
          if (fbox.mode == 0) break;
          init_points(fbox.x, fbox.y, fbox.dx, fbox.dy);
          fbox.mode = 0;
          break;
        case 'z': /* Zoom in on boxed region */
        case 'Z':
          if (fbox.mode == 0) break;
          xmin = vx(fbox.x);
          ymin = vy(fbox.y+fbox.dy);
          ymax = vy(fbox.y);
          xmax = vx(fbox.x+fbox.dx);
          set_range(xmin, ymin, xmax, ymax);
          init_points(0, 0, vp_width(), vp_height());
          pc_textmode();        /* Kluge to clear screen */
          init_video(EGA);
          fbox.mode = fbox.on = 0;
          break;
        case '1':
        case '2':
        case '3':
        case '4':
        case '5':
        case '6':
        case '7':
        case '8':
        case '9':
          dm = c-'0';
          break;
        case 0:
          c = getch();
          new_box = fbox;
          if (fbox.mode == 1) { /* moving the box */
            switch (c) {
              case EX_UP:
                new_box.y -= dm;
                break;
              case EX_DOWN:
                new_box.y += dm;
                break;
              case EX_RIGHT:
                new_box.x += dm;
                break;
              case EX_LEFT:
                new_box.x -= dm;
                break;
              default: break;
            }
          } else if (fbox.mode == 2) {
            switch (c) {
              case EX_UP:
                new_box.dy -= dm;
                break;
              case EX_DOWN:
                new_box.dy += dm;
                break;
              case EX_RIGHT:
                new_box.dx += dm;
                break;
              case EX_LEFT:
                new_box.dx -= dm;
                break;
              default: break;
            }
          }
          check_box(&new_box);
          break;
        default:
          break;
      }
    }
    if (fbox.mode == 0) break;
    else if (!fbox.on) flip_box();
  }
  if (fbox.on) flip_box();
}

/* Generate cycles through all the pixels, possibly starting over as
   dictated by menu().  Generate never returns.
*/
void generate(void) {
  int c, x, y;
  double fx, fy;

  for (;;) {
    while (next_point(&x, &y) == 0) {
      fx = vx(x);
      fy = vy(y);
      c = mandel_color(fx, fy);
      set_color1(c);
      draw_point(x, y);
      if (kbhit()) menu();
    }
    menu();
  }
}

void main(int argc, char **argv) {
  if (init_video (EGA)) {
    init_points(0, 0, vp_width(), vp_height());
    set_range(-2., -.95, .75, .95);
    generate();
  } else printf("Cannot select graphics mode");
}
```

End Listing Three

Listing Four

```
/* points.h include file for pixel ordering program. */

void init_points(int left, int top, int width, int height);

int next_point(int *dx, int *dy);

int mandel_color(double cx, double cy);

void set_range(double xmin, double ymin, double xmax, double ymax);

double vx (int dx);             /* convert device x to virtual x */

double vy (int dy);             /* convert device y to virtual y */
```

End Listing Four

Listing Five

```
To be included in grafix.h


/* Added by Norton Allen */
/* -------------------- */
void set_write_mode(int mode);

#define WM_REPLACE 0
#define WM_XOR 0x18
```

End Listing Five

Listing Six

```
/* wmode.c for DDJ grafix library Copyright (c) 1989 by Norton T. Allen */

#include <dos.h>
#include "grafix.h"

int write_mode = WM_REPLACE;

void set_write_mode(int mode) {
  write_mode = mode;
}
```

End Listings

Listing One *(Text begins on page 62.)*

```
;       protmode.asm -- 286 protected-mode instructions
;       requires MASM 5.0 or higher or TASM
;       masm -ml protmode;
;       or, tasm -ml protmode;

        dosseg

        .286p
        .model large
        .code

        public  _lsl, _lar, _verr, _verw, _sgdt, _sidt, _sldt

;       extern unsigned far lsl(unsigned short sel);
;       input:   selector
;       output:  if valid and visible at current protection level,
;                   return segment limit (which is 0 for 1-byte seg!)
;                else
;                   return 0
;
_lsl    proc
        enter   0, 0
        sub     ax, ax
        lsl     ax, [bp+6]
        leave
        ret
_lsl    endp

;       extern unsigned short far lar(unsigned short sel);
;       input:   selector
;       output:  if valid and visible at current protection level,
;                   return access rights (which will never be 0)
;                else
;                   return 0
;        bug:   retval for 1-byte seg = retval for invalid selector
;
_lar    proc
        enter   0, 0
        sub     ax, ax
        lar     ax, [bp+6]
        shr     ax, 8
        leave
        ret
_lar    endp

;       extern BOOL far verr(unsigned short sel);
;       input:   selector
;       output:  valid for reading ? 1 : 0
;
_verr   proc
        enter   0, 0
        mov     ax, 1
        verr    word ptr [bp+6]
        je      short verr_okay
        dec     ax
verr_okay:
        leave
        ret
_verr   endp

;       extern BOOL far verw(unsigned short sel);
;       input:   selector
;       output:  valid for writing ? 1 : 0
;
_verw   proc
        enter   0, 0
        mov     ax, 1
        verw    word ptr [bp+6]
        je      short verw_okay
        dec     ax
verw_okay:
        leave
        ret
_verw   endp

;       extern void far sgdt(void far *gdt);
;       input:   far ptr to 6-byte structure
;       output:  fills structure with GDTR
;
_sgdt   proc
        enter 0, 0
        les   bx, dword ptr [bp+6]
        sgdt  fword ptr es:[bx]
        leave
        ret
_sgdt   endp

;       extern void far sidt(void far *idt);
;       input:   far ptr to 6-byte structure
;       output:  fills structure with IDTR
;
_sidt   proc
        enter 0, 0
        les   bx, dword ptr [bp+6]
        sidt  fword ptr es:[bx]
        leave
        ret
_sidt   endp
;
;       extern unsigned short sldt(void);
;       input:   none
;       output:  Local Descriptor Table register (LDTR)
;
_sldt   proc
        sldt  ax
        ret
_sldt   endp

        end
```

End Listing One

Listing Two

```
/* PROTMODE.H */

typedef enum { FALSE, TRUE } BOOL;
#ifdef InstantC
unsigned far lsl(unsigned short sel)        {extern;}
unsigned short far lar(unsigned short sel)  {extern;}
BOOL far verr(unsigned short sel)           {extern;}
BOOL far verw(unsigned short sel)           {extern;}
void far sgdt(void far *gdt)                {extern;}
void far sidt(void far *idt)                {extern;}
unsigned short sldt(void)                   {extern;}
#else
extern unsigned far lsl(unsigned short sel);
extern unsigned short far lar(unsigned short sel);
extern BOOL far verr(unsigned short sel);
extern BOOL far verw(unsigned short sel);
extern void far sgdt(void far *gdt);
extern void far sidt(void far *idt);
extern unsigned short sldt(void);
#endif
```

End Listing Two

Listing Three

```
/* BROWSE.C */

#ifdef InstantC
#loadobj "protmode.obj"
#endif
#include "protmode.h"

void browse()
{
    unsigned long addr;
    unsigned i, acc;
    for (i=0; i<0xFFFF; i++)        // for all possible selectors
        if (acc = lar(i))           // if a valid selector
        {
            addr = D16AbsAddress(MK_FP(i,0));
            printf("%04X %061X LAR=%02X LSL=%04X PL=%02X %s %s %s %s\n",
                i,                              // selector
                addr,                           // physical base addr
                acc,                            // access-rights byte
                lsl(i),                         // segment limit
                i & 3,                          // protection level
                verr(i) ? "VERR" : "    ",      // readable?
                verw(i) ? "VERW" : "    ",      // writeable?
                i & 4 ? "LDT" : "GDT",          // which table?
                i == addr >> 4 ? "TRANS" : ""); // transparent?
        }
}
```

End Listing Three

Listing Four

```
/* SEL.C */

void sel(void far *fp)
{
    extern DESCRIPTOR far *gdt;
    extern DESCRIPTOR far *ldt;
    unsigned seg = FP_SEG(fp);
    unsigned index = seg >> 3;
    DESCRIPTOR far *dt = (seg & 4) ? gdt : ldt; // table indicator
    ACCESS_RIGHTS *pacc = (ACCESS_RIGHTS *) &dt[index].access;
    printf("SEL=%04X ADDR=%02X%04X LIMIT=%04X ACCESS=%d%c%c%c%c%c\n",
        seg, dt[index].addr_hi, dt[index].addr_lo, dt[index].limit,
        // display access rights as if they were file attributes:
        pacc->dpl,
        pacc->accessed ? 'a' : '-',
        pacc->read_write ? ((pacc->code_data) ? 'r' : 'w') : '-',
        pacc->conf_exp ? ((pacc->code_data) ? 'f' : 'e') : '-',
        pacc->code_data ? 'c' : 'd',
        pacc->xsystem ? '-' : 's',
        pacc->present ? 'p' : '-');
}
```

End Listings

Listing One *(Text begins on page 78.)*

```
#include <dos.h>        /* i.e. just like MS C */
void note();
main()
{
        note(440,500);
}
void note(pitch, duration)
int pitch,duration;
{
           int u,v;
        union REGS regs;
        unsigned int_count,int_duration,count,int_pitch;
           int_pitch = 1190000/pitch;
        int_duration = (duration*1821)/10000;
        regs.x.ax = 0;                          /* call timer */
        int86(0x1a, &regs, &regs);
        int_count = regs.x.dx;  /* internal count = lowest 16-bits of time*/
        u = inp(0x61) | 3;    /* Turn on channel 2 of 8255 using port 61h */
        outp(0x61,u);         /* send byte to back */
        outp(0x43,0xb6);      /* set up I/O register */
        outp(0x42,(char) int_pitch);    /* send freq to latch */
        outp(0x42,(int_pitch >> 8));
        do {
                regs.x.ax = 0;       /* use timer to get end of duration */
                int86(0x1a, &regs, &regs);
                count = regs.x.dx;   /* use lowest 16-bits of count */
        } while (count < int_duration + int_count);
        v = inp(0x61) & 0xfc;        /* turn off the sound */
        outp(0x61,v);
}
```

End Listing One

Listing Two

```
#define outp(p,v) dx = p;al = v;asm(dx,al,"     out     dx,al")
#define inp(p,v)  dx = p;asm(dx,"       in      al,dx",al);v = al
        unsigned char x,y,page = 0; /* globals for pc_test */
void note();
main()
{
        note(440,500);
}

void note(pitch, duration)
int pitch,duration;
{
        reg$eax unsigned short ax;
        reg$eax unsigned char al,x,y;
        reg$edx unsigned short dx;
        reg$ah  unsigned char ah;
        /* this section was added for play */
    unsigned int_count,int_duration,count,int_pitch;
        if (duration == 0) return;
        int_duration = (duration*1821)/10000;
/*      We left the original interrupt as a comment for comparison purposes */
/*      regs.x.ax = 0;       call timer */
/*      int86(0x1a, &regs, &regs);  */
/*      int_count = regs.x.dx; internal count = lowest 16-bits of time*/
/*      the inline assembly language is line for line identical in function
        although there is an obvious difference in format. */
        ax = 0;
        asm(ax,"        int     0lah",dx);
        int_count = dx;

        if (pitch==0) goto time_it;
        int_pitch = 1190000/pitch;
/*      The port input is a little different using inline asm macros */
/*      x = inp(0x61) | 3;      the original code becomes */
        inp(0x61,x);    /* Turn on channel 2 of 8253 using port 61H */
        x = x | 3;      /* After read turn on lowest 2 bits */
/*      The outp macro looks just like the outp function */
           outp(0x61,x);          /* send byte to back */
        outp(0x43,0xb6);       /* set up I/O register */
        outp(0x42,(char) int_pitch);    /* send freq to latch */
        outp(0x42,(int_pitch >> 8));
time_it:
        do {
                ax = 0;         /* use timer to wait for end */
                asm(ax,"        int     0lah",dx);
                count = dx;
        } while (count < int_duration + int_count);
        inp(0x61,y);    /* Turn off channel 2 */
        y = y & 0xfc;   /* use 1111 1100 to turn off lowest 2 bits only */
        outp(0x61,y);
}
```

End Listing Two

Listing Three

```
        name sound4.c
        .387
        assume  cs:codeseg
        assume  ds:dataseg
codeseg segment dword er use32 public 'code'
dataseg segment dword rw use32 public 'data'

dataseg ends
        align   4
_note   proc    near

        push            edi
        push            esi
        push            ebx
        mov             ebx,[esp]+16
        cmp             dword ptr [esp]+20,0
        jne             L17     short
        pop             ebx
        pop             esi
        pop             edi
        ret
        align   4
L17:
        mov             ecx,10000
        imul            eax,[esp]+20,1821
        cdq
        idiv            ecx
        mov             edi,eax
        mov             ax,0
        int             0lah
        movzx   esi,dx
        or              ebx,ebx
        jne             L16     short
        jmp             L13
        align   4
L16:
        mov             eax,1190000
        cdq
        idiv            ebx
        mov             ebx,eax
        mov             dx,97
        in              al,dx
        or              al,3
        mov             dx,97
        out             dx,al
        mov             dx,67
        mov             al,182
        out             dx,al
        mov             dx,66
        mov             al,bl
        out             dx,al
        mov             dx,66
        mov             eax,ebx
        shr             eax,byte ptr 8
        out             dx,al
        align   4
L14:
        align   4
L13:
        mov             ax,0
        int             0lah
        movzx   ecx,dx
        mov             eax,edi
        add             eax,esi
        cmp             eax,ecx
        ja              L13     short
        mov             dx,97
        in              al,dx
        and             al,252
        mov             dx,97
        out             dx,al
        align   4
L9:
        pop             ebx
        pop             esi
        pop             edi
        ret
        align   4
_note   endp
dataseg segment dword rw use32 public 'data'

;_ax                    ax              local
;_al                    al              local
;_x                     al              local
;_y                     al              local
;_dx                    dx              local
;_ah                    ah              local
;_int_count     esi             local
;_int_duration  edi             local
;_count         ecx             local
;_int_pitch     ebx             local

;parameters
;_pitch         ebx             local
;_duration      [esp]+20        local
dataseg ends
        end
```

End Listing Three

Listing Four

```
#include <stdio.h>
#include <dos1.h>
#include <ctype.h>
/*      ***WARNING*** if you change the scale so that it starts
        on middle C, instead of A, the resulting routine will
        exhibit not only the look and feel of the BASIC PLAY
        command, but its sound as well.
*/
#define aa      440             /* middle a = 440 */
#define as      469             /* a sharp */
#define bb      493
#define cc      523             /* middle c */
#define cs      556
#define dd      587
#define ds      624
```

(Listing continued on page 124)

Listing Four *(Listing continued, text begins on page 78.)*

```
#define ee      659
#define ff      698
#define fs      739
#define gg      783
#define gs      832

#define outp(p,v) dx = p;al = v;asm(dx,al,"     out     dx,al")
#define inp(p,v)  dx = p;asm(dx,"       in      al,dx",al);v = al
        unsigned char x,y,page = 0; /* globals for pc_test */
void note();
void look_ahead_and_toot();
void play();
int check_length();
int check_integer();
int gobble_dots();
                /* the buffer for the notes to be input */
int  pitch;
int  count = 0;                 /* points to current location in string */
int  length = 4;                        /* default is a quarter note */
int  tempo = 240;               /* = 120 beats per minute */
int  duration = 60;             /* = tempo/length =1/4 @ 120 bpm */
int  shift = 0;                 /* current octave shift factor */
char c_note;                    /* the current note character used for diag */

main()
{       /* Stereo version of Heart and Soul for two PCs
        lifted from a BASIC program Transcribed by
        Michael Benjamin Fried - Age 11 */
        /* bass line plays on first machine */
        play( "T150L8O4CCEEAACCDDFFO3GG>BBO4");
        play( "L8O4CCEEAACCDDFFO3GG>BBO4");

        /* while melody plays on a second */
        play( "O4L4CCC.P32C8B8A8B8C8D8P32");
        play( "EEE.P32E8D8C8D8E8F8G.C.>A8<G8F8E8D8CB8Ao3G8FFGGo4");
}
void play(in_string)
char in_string[];
{
int temp_duration;              /* gets set by L or change in l */
int temp_octave;                /* holds temporary octave */
char    n_note;
count = 0;
printf("note = %s \n",in_string);
while (c_note = in_string[count]){ /* loop till out of characters */
        n_note = in_string[count+1];  /* look ahead 1 char now */
        switch(c_note){                 /* switch on current note */

                case 'A':               /* do a,a sharp and a flat */
                case 'a':
                        pitch = aa;     /* set the default to A natural */
                        if ((n_note == '#')||(n_note == '+')){
                                pitch = as; /* it was A sharp */
                                count++;
                                }
                        if (n_note == '-'){ /* it was A flat */
                                pitch = gs; /* A flat == G sharp */
                                count++;
                                }
                        look_ahead_and_toot(in_string); /* self explanatory */
                        break;                          /* line duration */

                case 'B':               /* B is just like A */
                case 'b':
                        pitch = bb;
                        if ((n_note == '#')||(n_note == '+')){
                                pitch = cc; /* B sharp is actually C */
                                count++;
                                }
                        if (n_note == '-'){
                                pitch = as; /* B flat is A sharp */
                                count++;
                                }
                        look_ahead_and_toot(in_string);
                        break;

                case 'C':                       /* C is just like A */
                case 'c':
                        pitch = cc;
                        if ((n_note == '#')||(n_note == '+')){
                                pitch = cs;
                                count++;
                                }
                        if (n_note == '-'){ /* C flat is actually B */
                                pitch = bb; /* and a perfectly legal note */
                                count++;
                                }
                        look_ahead_and_toot(in_string);
                        break;

                case 'D':               /* D is like A */
                case 'd':
                        pitch = dd;
                        if ((n_note == '#')||(n_note == '+')){
                                pitch = ds;
                                count++;
                                }
                        if (n_note == '-'){
                                pitch = cs; /* D flat is C sharp */
                                count++;
                                }
                        look_ahead_and_toot(in_string);
                        break;

                case 'E':               /* E is like A */
                case 'e':
                        pitch = ee;
                        if ((n_note == '#')||(n_note == '+')){
                                pitch = ff; /* E sharp is F */
                                count++;
                                }
                        if (n_note == '-'){
                                pitch = ds;
                                count++;
                                }
                        look_ahead_and_toot(in_string);
                        break;

                case 'F':               /* F is like A */
                case 'f':
                        pitch = ff;
                        if ((n_note == '#')||(n_note == '+')){
                                pitch = fs;
                                count++;
                                }
                        if (n_note == '-'){
                                pitch = ee;
                                count++;
                                }
                        look_ahead_and_toot(in_string);
                        break;

                case 'G':               /* G is like A */
                case 'g':
                        pitch = gg;
                        if ((n_note == '#')||(pitch == '+')){
                                pitch = gs;
                                count++;
                                }
                        if (n_note == '-'){
                                pitch = fs;
                                count++;
                                }
                        look_ahead_and_toot(in_string);
                        break;

                case 'L':               /* set length */
                case 'l':
                        if(temp_duration = check_length(in_string)){
                                duration = tempo/temp_duration;
                                length = temp_duration;
                                }
                        break;
                case '>':               /* go up an octave */
                        shift++;
                        break;

                case '<':               /* go down an octave */
                        shift--;
                        break;

                case 'O':               /* chose an octave */
                case 'o':
                        temp_octave = n_note - '0';
                        if ((temp_octave < 0)||(temp_octave > 6)){
                                printf("octave out of range");
                                break;
                                }
                        switch(n_note){
                                case '0':
                                        shift = -4;
                                        break;
                                case '1':
                                        shift = -3;
                                        break;
                                case '2':
                                        shift = -2;
                                        break;
                                case '3':
                                        shift = -1;
                                        break;
                                case '4':               /* default octave */
                                        shift = 0;
                                        break;
                                case '5':
                                        shift = 1;
                                        break;
                                case '6':
                                        shift = 2;
                                        break;
                                }
                        count++;        /* advance over digit */
                        break;

                case 'P':               /* set pause/rest length */
                case 'p':               /* issue note of freq 0 to rest */
                case 'R':               /* computer scientists pause */
                case 'r':               /* but musicians rest! */
                        pitch = 0;
                        look_ahead_and_toot(in_string);
                        break;

                case 'T':               /* set tempo */
                case 't':
                        temp_duration = check_integer(in_string);
                        if ((temp_duration < 32)||(temp_duration > 255))
                                break;
                        tempo = temp_duration*2;
                        duration = tempo/length;
                        break;

                case ' ':               /* spaces are gobbled up */
                        break;

                default:                /* had a problem so issue error */
                        printf("Syntax error in character %d \n",count);
                        goto terminate;
                }
        count++;        /* advance pointer to next note */
}
```

```
terminate:
}
/* The trickiest part of the syntax are the optional trailers that
        can follow each note. These include an optional integer that
        specifies a quarter (4) or eighth note (8) (or any integer
        between 1 and 64) and 1 or more optional dots, each of which
        increases the current duration by half. This section parses
        these trailers, and then uses the global variables that contain
        the tempo and octave to compute the duration and pitch, and
        then call note. Note that rests are handled as notes of 0 pitch.
*/

void look_ahead_and_toot(in_string)
char in_string[];
{
int temp_duration;
if (temp_duration = check_length(in_string)) /* if non zero have a temp */
        temp_duration = tempo/temp_duration;  /* compute new duration */
else
        temp_duration = duration;        /* if 0 play default duration */
/* check for dot, and if found call gobble_dots to increase temp_duration */
if (in_string[count+1] == '.')
        temp_duration = gobble_dots(temp_duration,in_string);
                /* range check octaves */
if (shift < -4)
        shift = -4;
if (shift > 2)
        shift = 2;
                /* shift to change octaves */
if (shift < 0)  /* negative shifts go down in frequency */
        pitch = pitch >> -shift;
else            /* positive shifts go up in frequency */
        pitch = pitch << shift;
        /* optional diagnostics for debugging */
printf("%c = %d duration = %d octave = d\n",
                        c_note,pitch,temp_duration,shift+4);
        /* finally we are ready for a little toot */
note(pitch,temp_duration);
}
int gobble_dots(duration_in,in_string)
int duration_in;
char in_string[];
{
int duration_out;
int duration_increment;
duration_out = duration_in;
duration_increment = duration_in;
/* gobble as long as there are dots adding half the prior duration inc */
while (in_string[count+1] == '.'){
        duration_increment = duration_increment >> 1; /* divide it by 2 */
        duration_out = duration_out + duration_increment;
        count++;                        /* advance string pointer */
        }
return(duration_out);
}
/* returns 1-64 in range 1-64 else returns 0 */
int check_length(in_string)
char in_string[];
{
int result = check_integer(in_string);
        if ((result < 1) || (result > 64))
                return(0);      /* out of range */
        else
                return(result); /* in range */
}
/* 0 1 to 999 if 1 - 999 found and advances count */
/* returns 0 otherwise */
int check_integer(in_string)
char in_string[];
{
int n_char,m_char,l_char;
n_char = in_string[count+1] - '0';
if ((n_char > 9) || (n_char < 0)) return (0); /* return if out of range */
count++;                        /* we found a digit so advance count */
m_char = in_string[count+1] - '0';   /* check next integer */
if ((m_char > 9) || (m_char < 0))
        return (n_char);
else
        count++;                /*found second didit so advance again */
l_char = in_string[count+1] - '0';
if ((l_char > 9) || (l_char < 0)) /* check last possible digit */
        return (n_char*10+m_char); /* compute 2 digit result */
else {
        count++;                /* we found a third and last digit */
        return(n_char*100+m_char*10+l_char);
        }
}
/* optional main program works as an interactive interpreter */
char note_string[120];
/*
main()
{
do{
        puts("enter note ");
        fflush(stdin);
        gets(note_string);
        printf("\nnote string = %s \n",note_string);
        play();
        } while (strlen(note_string) > 0);
}
*/
```

End Listings

LZW Revisited

Speeding up an old data compression favorite

Shawn M. Regan

When the October 1989 issue of *Dr. Dobb's* came out, I was delighted to see Mark Nelson's article on LZW data compression ("LZW Data Compression"). Mark presented a clear description with code of a basic LZW compression program. As I used the program, however, I discovered that for some larger files, even when using 14-bit codes, the compressed file would actually be larger than the original. Because Mark's intention was to enlighten and not to complicate the subject, his code omitted some optimizing additions to LZW compression. Although, he did describe some optimization techniques — including the use of variable code size — as well as clearing the string table after the compression ratio degrades.

The original program's lack of performance on larger files can be traced to the fixed-length string table. When the table is full, new codes cannot be added and must be sent out in character form with no compression done. When this happens, you could actually be sending out an 8-bit character using a 14-bit code; this explains how the file can get larger. If your incoming data changes, even moderately, with the string table full, then the compression ratio begins to degrade rapidly.

With this in mind, imagine compressing a small file with your code size set at 14 bits. If your string table needs less than 511 entries, you could have used 9-bit codes saving 5 bits per output code. Of course you wouldn't want to fix your code size at 9 bits because the compression on larger files would suffer. What you would like is for the code size to start at 9 bits and if that table filled, the code size could increment to 10, thus providing optimal performance on any size file.

Shawn is a programmer/analyst for MicroBilt Inc. of Atlanta, Georgia. You can reach him through Interlink; or write to him at 2127B Powers Ferry Rd., Marietta, GA 30067.

My Implementation

One of the more elegant features about LZW compression are that the compression and expansion programs build the exact same string table for a particular file. This means at the same point the compression program's string table is full so is the expansion program's. At this point you should try to adjust your code size. As you can see in the compression section of Listing One (page 127), I wait until after I have sent out the current code before the code size is incremented because the current code belongs to the previous code size. Notice in the compression section I increment when the code size is greater than *max_code*, while in the expansion program I increment when the code size equals *max_code*. This is because the expansion section is working a code behind using *old_code* instead of *new_code*. It is also because of this that I must handle a special case when incrementing the code size on an end-of-file condition.

Finally, you should also set some arbitrary limit on your code size. If you use 14 bits, your codes stay well under the positive integer maximum of 32767, which suits the program without any modification. Don't forget your table size needs to be a prime number somewhat larger than 2^MAX_CODE_SIZE. To implement the table clearing, start by monitoring the number of bytes (not codes) read in and then sent out. After a predetermined interval, compute the new compression ratio and check it against the previous one. If the ratio has increased, you need to clear out the string table and start over. You then need a device to send a signal from the compression program to the expansion program to clear the string table. The easiest way to accomplish this is by reserving the first of the 9-bit codes. In my example, I used 256 as the *CLEAR_TABLE* code. I also used 257 as the *TERMINATOR* to signal the end-of-file condition. This means the first available code for compression is now 258, which I've defined as *FIRST_CODE*.

When combining both methods, you should not experience any degradation in compression until the table is full. When the table is full, you will first check to see if you can increase the code size. If you can't, then (and only then) will you start to monitor your compression ratio at your predefined interval and ultimately clear the string table. When this happens, you can reset your code size back to 9-bits because basically you're starting from scratch. Although you still won't get performance as good as PKZIP (from

Before	After	% of Original	File type
115,094	40,636	35 %	.C - C program
11,054	4,811	43 %	.C - C program
230,582	141,659	61 %	.EXE - Executable
16,944	12,905	76 %	.EXE - Executable
90,610	20,806	22 %	.TXT - Redundant text file
110,592	64,804	58 %	.LIB - C object library

Table 1: *Typical compression levels using revised LZW*

PKWARE, Glendale, Wisc.), you now have the source for a much improved version of this basic LZW compression program. Table 1 lists some typical compression levels I've achieved with this program.

Your Implementation

Even though the program works well as is, there are still some improvements that can be made. As Mark suggested, the input and output routines can be

For better compression, you might experiment with table clearing

modified for more speed. Also, a more sophisticated hashing routine might speed it up. For better compression, you might experiment with table clearing. I found on .EXE files the compression ratio drops steadily after a code size increase, then bottoms out and then starts rising again. If you suspend clearing until you are back to just below the starting ratio you can get a somewhat better compression. I also noticed that in smaller text files, I can at times get better compression by clearing the table instead of increasing the code size. Be careful, however, about basing any optimization methods on any preanalysis of the data. If, for example, you wish to use it with stream I/O you will be working with buffers and not files where any preanalysis might be difficult or useless.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

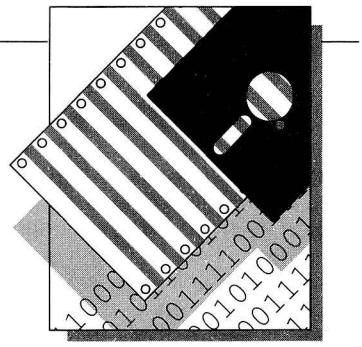
DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 12.**

Listing One

```
/* Basic LZW Data Compression program published in DDJ October 1989 issue.
 * Original Author: Mark R. Nelson
 * Updated by: Shawn M. Regan, January 1990
 * Added: - Method to clear table when compression ratio degrades
 *        - Self adjusting code size capability (up to 14 bits)
 * Updated functions are marked with "MODIFIED". main() has been updated also
 * Compile with -ml (large model) for MAX_BITS == 14 only
 */

#include <stdio.h>

#define INIT_BITS 9
#define MAX_BITS 14             /* Do not exceed 14 with this program */
#define HASHING_SHIFT MAX_BITS - 8

#if MAX_BITS == 14              /* Set the table size. Must be a prime    */
#define TABLE_SIZE 18041        /* number somewhat larger than 2^MAX_BITS.*/
#elif MAX_BITS == 13
#define TABLE_SIZE 9029
#else
#define TABLE_SIZE 5021
#endif

#define CLEAR_TABLE 256    /* Code to flush the string table */
#define TERMINATOR  257    /* To mark EOF Condition, instead of MAX_VALUE */
#define FIRST_CODE  258    /* First available code for code_value table */
#define CHECK_TIME  100    /* Check comp ratio every CHECK_TIME chars input */

#define MAXVAL(n) (( 1 <<( n )) -1)   /* max_value formula macro */

unsigned input_code();
void *malloc();

int *code_value;                      /* This is the code value array */
unsigned int *prefix_code;            /* This array holds the prefix codes */
unsigned char *append_character;      /* This array holds the appended chars */
unsigned char decode_stack[4000];     /* This array holds the decoded string */

int num_bits=INIT_BITS;               /* Starting with 9 bit codes */
unsigned long bytes_in=0,bytes_out=0; /* Used to monitor compression ratio */
int max_code;                         /* old MAX_CODE */
unsigned long checkpoint=CHECK_TIME;  /* For compression ratio monitoring */

main(int argc, char *argv[])
{
   FILE *input_file, *output_file, *lzw_file;
   char input_file_name[81];
 /* The three buffers for the compression phase.  */
   code_value=malloc(TABLE_SIZE*sizeof(unsigned int));
   prefix_code=malloc(TABLE_SIZE*sizeof(unsigned int));
   append_character=malloc(TABLE_SIZE*sizeof(unsigned char));

   if (code_value==NULL || prefix_code==NULL || append_character==NULL) {
      printf("Error allocating table space!\n");
      exit(1);
   }
 /* Get the file name, open it, and open the LZW output file. */
   if (argc>1)
      strcpy(input_file_name,argv[1]);
   else {
      printf("Input file name: ");
      scanf("%s",input_file_name);
   }
   input_file=fopen(input_file_name,"rb");
   lzw_file=fopen("test.lzw","wb");
   if (input_file == NULL || lzw_file == NULL) {
      printf("Error opening files\n");
      exit(1);
   }
   max_code = MAXVAL(num_bits);     /* Initialize max_value & max_code */
   compress(input_file,lzw_file);       /* Call compression routine */

   fclose(input_file);
   fclose(lzw_file);
   free(code_value);                    /* Needed only for compression */

   lzw_file=fopen("test.lzw","rb");
   output_file=fopen("test.out","wb");
   if (lzw_file == NULL || output_file == NULL) {
      printf("Error opening files\n");
      exit(1);
   }
   num_bits=INIT_BITS;                  /* Re-initialize for expansion */
   max_code = MAXVAL(num_bits);
   expand(lzw_file,output_file);        /* Call expansion routine */

   fclose(lzw_file);                    /* Clean it all up */
   fclose(output_file);
   free(prefix_code);
   free(append_character);
}
/* MODIFIED This is the new compression routine. The first two 9-bit codes
 * have been reserved for communication between the compressor and expander.
 */
compress(FILE *input, FILE *output)
{
   unsigned int next_code=FIRST_CODE;
   unsigned int character;
   unsigned int string_code;
   unsigned int index;
   int i,             /* All purpose integer */
   ratio_new,         /* New compression ratio as a percentage */
   ratio_old=100;     /* Original ratio at 100% */

   for (i=0;i<TABLE_SIZE;i++)   /* Initialize the string table first */
      code_value[i]=-1;
   printf("Compressing\n");
   string_code=getc(input);     /* Get the first code */
```

(continued on page 167)



HyperCard and (or?) Hypertext

Michael Swaine

The only honest way to start a column about the suitability of this product for hypertext is by quoting its creator:

"I don't think of HyperCard at all as a hypertext system. It is not a very good one." — Bill Atkinson.

That would also be the end of the column if every father knew his own child. But out there on the nets over the past three years, HyperCard has been doing things Bill never envisioned, and is playing an important role in the development of hypertext systems. The role is a research role.

Field Research

Since Apple caved in to Atkinson's ultimatum, agreed that HyperCard was system software, and began bundling it with every new Macintosh back in 1987, HyperCard authors of varying levels of skill have been all over the nets, showing their work with the obnoxious eagerness of a teacher's pet in grade school. I predicted in *DDJ* back then that the release of HyperCard would lead to a hyperglut of trashware, a pretty obvious prediction.

Not all this hyperactivity is trashware; and even in the trash there are some interesting items. (Jean-Louis Gasseé says that this latter point is not lost on trash can grubbing "MacLeak" reporters, but he's just grumpy because he had to give back the company car.) (Many Americans discovered the joy of searching through trash cans in the 1980s, for which Apple must share the credit with Ronald Reagan.) (The parenthetical note is one technique that writers use to overcome the limitations of linear text. It's a kludge.) Interesting items, I was saying, thinking particularly of what can be learned from that "interesting experiment" that didn't quite work out. A lot of HyperCard development fits this description.

I want to call this undisciplined, uncontrolled, enthusiastic amateur HyperCard development work "research," but this is bending the word. Then again, magazines sometimes use the word "research" to refer to casual reader-interest surveys, which is just as bad. These two uses are, and you saw this word here first, "isotropes" — they're similarly bent.

Moving right along. There is a lot of serious, professional, controlled experimentation in hypertext going on in universities and in the few corporate research labs that haven't turned into short-range product development shops. In particular, the people at Brown University's IRIS (Institute for Research in Information and Scholarship) have been following one thread of hypertext research ever since Ted Nelson coined the term, and have demonstrated the virtues of hypertext for promoting nonlinear thinking. This kind of result does not come from playing with HyperCard and uploading your work to CompuServe.

But in a looser sense, these HyperCard experiments, failed and flawed as they may be, are crude offerings of aspects of hypertext. Seeing what has been tried, and how it has been received out there among the savages, is an education in hypertext field research.

"Crude offerings," I say, but I do not mean it as a criticism of the developers. HyperCard is, as Atkinson acknowledges, not a hypertext system. What is it, and why are people using it to do hypertext-like things? Atkinson calls it a "software erector set." That's pretty accurate, down to the point made by one critic that when you build things with an erector set, you get things that look like they were built with an erector set. HyperCard provides a limited set of classes of objects: Stacks (which are HyperCard documents), cards (information chunks displayed as screen-sized bitmap), fields for text (which can exceed the screen size by scrolling), and buttons (basically icons with attached programs). In limited object-oriented style, HyperCard lets you create stacks by writing message handlers and associating them with instances of these classes of objects. The basic metaphor is of a stack of 3 × 5 cards, each card displaying text and/or graphics, and the cards linked to one another quite freely, with buttons being the usual tool for following the links.

This model supplies several things that a hypertext system ought to have. The user can create links among chunks of information without doing any programming and the programmer can create different links relatively easily. HyperTalk, the programming language in HyperCard, is interpreted, so it's a quick development tool, yet it's powerful for an interpreted language. Apple gives HyperCard away with every new machine, so you can count on Mac users having it, and this also has led to a lot of people having fiddled with it, so HyperCard is helping to spread familiarity with some hypertext ideas.

It also lacks some things a hypertext system needs. It has no built-in navigation system, although Apple encourages stack developers to include maps in their stacks. There is nothing analogous to a browser. HyperCard's links are coarse-grain links, taking the reader from card to card. Fine-grain linking, triggering off items the size of an individual word, is possible, but is not inherent in the product. HyperCard has no text links.

In Search of the Missing Link

The biggest failing of HyperCard for anyone interested in hypertext is the lack of text links. By the time this column sees print — I'm writing this in January, believe it or not — Apple should have released Version 2.0 of HyperCard, widely expected to implement some kind of text links. I think it's fairly safe to predict that Apple's text links will not satisfy all those people who want HyperCard to be a better hypertext tool.

The text link solution supported in Version 2 is likely to be an extension of the button model. As I write this, buttons are the chief tool in HyperCard for supporting links. A button is restricted to one line (usually a word or two) of text. Fields are where you put text in HyperCard. Fields were never designed to support linking.

HyperTalk programmers have been coding workarounds for this failing since HyperCard was released, though, and some of these are effective in limited domains. The work these programmers are doing with text links typifies the kind of work being done with HyperCard in other hypertext or hypermedia areas, and it shows how easy it is to test alternative hypertalk user interface techniques using HyperCard.

What Atkinson gave us is the button. To create a link to another section (card) of a HyperCard document (stack) or to another stack, you create a button and link it to the card or stack. The link is by default unidirectional, except that the user can always return from a linked card via a standard key combination or menu selection. If you want explicit bidirectional links, you have to create and link two buttons. (Buttons can also be used to trigger pop-up fields, for such fine-grained hypertext purposes as displaying, in a pop-up window or field, a definition of a term.)

The first thing many people did when they started working with HyperCard was to place an invisible button over a text field or a graphic to make something happen when the text or graphic was clicked on. The earliest stacks from Apple used this technique. One early third-party developer who made good use of such invisible buttons is Amanda Goodenough, whose AmandaStories stacks showed one way to author interactive, child-directed stories for young readers. But the buttons were constrained to be rectangles and were not really connected with the text or art with which they were associated. Change the text, copy the art and paste it elsewhere, move the button, and the button is no longer associated with the text or graphic.

Soon after HyperCard's release, Keith Rollin of Apple demonstrated how to write a HyperCard external command to implement region buttons. These buttons, which used QuickDraw regions, could take on arbitrary polygonal shapes. This development, along with the fixed bitmap of a card (which makes screen position a somewhat reasonable way to refer to a picture), appears to make transparent buttons fairly useful for linking off pictures. Good stacks continue to be developed using this technique. But it doesn't do the job for text: You can't even change the font in a field without having to reposition the button. For hypertext linking, you need to be able to link off text, not off a screen position.

One step toward better linking might be to extend the find facility of HyperCard. Users can already select text in a field and perform a find operation. Why not insert a step between selection of the text and the conventional search? If the text selected is on a list of hypertext link words, follow the link; otherwise, let the find proceed as usual. Harvey Chang and Steve Drazga are two stack developers who extended the *find* function. Listing One (page 153) shows Drazga's hypertext technique while Chang's is in Listing Two (page 153). Chang's technique starts by allowing the user to select any contiguous text, even dragging across word boundaries. Then it attempts to use the selected text as the name of a card in a "go to card . . . " command. If that fails, it uses the selected text as a conventional search string. His implementation suffers from the requirement that the user select text, then explicitly click on a hypertext button.

There are a couple of problems with this technique as an approach to hypertext. (Chang, I should point out, doesn't call it that.) First, it doesn't give the reader any indication of where the links are. Second, it raises the question of what unit of text can serve as links.

Which Text is Hyper?

Chang's technique doesn't identify the links, so the reader must hit them by chance. In other hypertext systems, the linking text is often identified by a change in type style: Boldface, italic,

underlining, or all of these, may be used. HyperCard doesn't make such an approach easy. Text fields in HyperCard are impoverished text of a single font, size, and style. One stack developer, Gregory Nelson, investigated what it would take to create even the appearance of rich text in HyperCard fields. He faked rich text by using 10 (!) overlapping fields and a thoroughly unworkable text-entry procedure. That blind alley is thoroughly mapped, thanks to Nelson.

John Anderson had better luck in introducing style variations in HyperCard fields, by creating a specialized font that included italic and roman characters. Because it means losing the special characters that had been in the font, this approach trades text impoverishment for font impoverishment, and is still a long way from support for the kind of hypertext link identification seen in products such as Guide.

HyperWords and HyperLines

The other problem with Chang's technique, viewed as an approach to hypertext linking, is that it depends on the select operation, which is nicely standardized across the Macintosh-user interface, but which is not a technique designed for grabbing links. This is particularly so in HyperCard, where the user can't select text unless the field is unlocked — in which case the text is subject to modification. You certainly don't want the user selecting a hypertext link only to delete it from the document by a careless touch of the delete key.

Many stackware developers have found the same fix for this problem. To allow the user to select a word in any field simply by clicking on it once as though it had a button attached, you do the following:

```
% lock the field
% wait for a click in the field
% save the coordinates
% unlock the field invisibly
% send two clicks to the coordinates
% save the selection
% relock the field
```

The first significant implementation of the single-word link I know of is XrefText by Frank Patrick (available as shareware from BMUG or BCS). Raines Cohen of Team BMUG inspired this shareware quasi-hypertext system, which lets users create links from arbitrary single words in fields by option-clicking, and lets them follow these links by simply clicking. XrefText limits its search for the selected word to a keyword field on every card. The link words are identified by an asterisk.

This works well for selecting a single

word, and I've found it a useful way to allow the user to select an item from a scrolling list, even when the items in the list are not single words. My technique, which handles moved or scrolled fields, is shown in Listing Three, page 153.

This technique is all right for picking a line from a field, but neither this nor the toggle-lock-and-simulate-double-click technique for picking a single word is something you want to be doing regularly in an interpreted language.

One serious problem with all these approaches is that not every key can be a single word or a complete line. Footnotes in linear text sometimes apply to a single word, but can also apply to a sentence, paragraph, chapter, proper name, or to the general idea of a passage. Hypertext links ought to be just as flexible.

From SuperScript to Tagsterisk, or Footnotes of the Future

John Anderson, when he was developing stacks under the Acme Dot logo, came up with something he called the "tagsterisk," a tool for identifying for the reader not only where links are in the text, but also what they are. "The idea," he said, "is to give the reader a clue indicating just where he or she will branch if he or she clicks on a tagged word." Tagsterisks are just alphabetic superscript characters to be used in the way that asterisks are used in linear text to point to footnotes. Anderson implemented them as part of his custom font.

Tagsterisks, he claimed, "give the reader a greater feeling of control over the hypertext process. The key is to use no more than three or four." One Anderson stack used four tagsterisks: *s* to flag a source reference, *m* for more information on the subject, *g* for a graphic, and *p* for pronunciation (the last two being hypermedia rather than hypertext). Another stack used *p* for a picture, *g* for a graph, *d* for a definition, and *l* to bring up a list of links to other documents on the subject. Only the last of these was actually implemented as a link to another card or stack; the other three tagsterisks triggered pop-up windows or fields.

Tagsterisks are not tied to any particular method of implementing the link, which could be done with a button over the text, a field script, or with whatever technique is included in Version 2 of HyperCard. It's worth noting that, if the linking script is associated with the tagsterisk character itself, it becomes possible to attach two or more links to one word, say, a picture and a definition, without ambiguity.

Hypertext has been called the extended footnote, but only as a reference to its closest analog in linear text, not as a design direction. Tagsterisks take the expression literally, extending the footnote in several dimensions. There are some advantages to this extended footnote approach over, say, boldfacing links in text. One could argue, for example, that hypertext ought to extend the concept of text, not limit it. Using boldface to represent links "uses up" this font style variation, precluding its use for other purposes. The proponent of boldface links could counter that every use of boldface in linear text is really a wannabe link. Maybe our typographic conventions and type style variations were developed precisely to circumvent the limitations of the linear medium. If so, boldface has been waiting all this time to be used for hypertext link identification. The response might be that it's a little naive to think that we haven't found other, nonhypertextual uses for these typographical conventions over the centuries.

It's not clear that either side of this argument is wrong. No doubt different techniques are appropriate for different purposes. Designers of training manuals, help systems, and the like seem to be doing all right with boldface and

italic variations to identify links. Universal hypertext may require something else, but universal hypertext *is* something else.

Whether or not one buys the argument that hypertext should extend text and not limit it, it seems clear that HyperCard should stop limiting its users' use of text. The impoverished text fields of HyperCard have not helped research in this area, either to investigate ways to use text style variations to identify links, or to show that this is a bad idea. HyperCard needs richer text fields.

Hypertext, though, is still very much in the research stage, and HyperCard stack development is one legitimate form of hypertext research. The laboratory studies, valuable as they may be, are not going to tell us much about the market acceptance of particular user-interface implementation decisions regarding hypertext. I recommend that anyone interested in hypertext download some stacks and take a look at them. Even better, upload some.

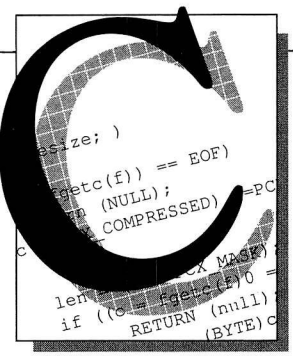
Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 153.)

Vote for your favorite feature/article.
Circle Reader Service **No. 9.**



HYPERTREE: A Hypertext Index Technique

Al Stevens

Programmers tend to think of hypertext in ways that we use it in our applications. A frequent use in our experience is in on-line help systems where the help text is organized into a structured set of help windows. You get to a help window as a result of some context-sensitive control, perhaps with the cursor on a keyword or the program at a particular place in a menu or data entry screen. The help system then allows you to select higher and lower levels of detail of help from keywords strategically positioned in the current window. Such help systems use lightweight hypertext concepts, but they are the ones we are most familiar with.

The real potential of hypertext technology comes when we consider using it for structured retrievals from large static text databases. Many disciplines deal regularly with such data. Lawyers, engineers, programmers, doctors, proposal writers, and researchers of all kinds work constantly with large amounts of boilerplate text. Hypertext offers a way to build structured and disciplined retrieval capabilities into these databases.

The heart of most heavy-duty hypertext applications is the index. To find your way to an associated body of text from a selected word or phrase, you need an index that translates the word into a pointer, into the text database. Last February we built an index system for our TEXTSRCH project that used a hashing technique to derive the pointer to a list of files where a selected keyword exists. That index technique was useful to the purpose of TEXTSRCH, which was the selection of files that match a Boolean keyword query. This application is a form of hypertext, but it too is a lightweight use of it.

This edition of the "C Programming" column looks at a different indexing technique, a loose adaptation of the the B-tree, one that offers a more heavy-duty kind of support to the problems of text searching. Besides providing the random selection of the hashed keyword, the B-tree allows you to navigate an index in the sequence of its keys as well. This facilitates the kind of search where a selected word delivers a list of words that are close, for example.

The B-tree is a data structure resembling an inverted tree. The root is at the top of the tree and the leaves are at the bottom. Each node contains key values, and each key value points to the node at the next lower level in the tree where values reside that are greater than it and less than the next adjacent key. The lower nodes are child nodes, the higher ones are parent nodes. Mixed metaphors, these families and trees. The pointers in the nodes at the bottom level — the leaves — are not node pointers but instead contain data values that match the keys. B-trees are typically used by database management systems to manage the indexes for primary and secondary data element keys. To support such applications, the traditional B-tree algorithms must be capable of adding keys in a random sequence and deleting them. I published the C language source code for such B-tree algorithms in two books, *C Development Tools for the IBM PC* (Brady Books, 1986) and *C Data Base Development* (MIS Press, 1987).

The modified B-trees that we will use to index a static hypertext database do not need to support key deletion, and the index construction can be done in the sequential order of the keys. These two features greatly simplify the code needed for tree maintenance. Because the index construction techniques are different, the trees of this method are not always precisely balanced after the fashion of B-trees. Every B-tree node below the root always contains at least half the number of keys that it can hold. This is because when a node is filled to capacity, it splits into two nodes to accommodate the next key. When it splits, the key value from the middle goes into the node above it in the tree. When a key is deleted, the node is combined with one of its sibling nodes if the two together can now hold the keys of both. The modified B-tree structure for this problem does not work that way. We add keys in sequential order, so the nodes fill up from left to right. No splitting occurs, and all but the rightmost node at each level will be full. Because the index supports a static database, no key deletion ever happens.

Our index is different from the B-tree in one other way. A B-tree consists of nodes each of which can contain a fixed number of keys. By definition, the key length is fixed if the node length is fixed. In our trees, the node length is fixed and the key length is variable.

We call this data structure the "HyperTree." It consists of a header record and some number of nodes. The header record contains the node number of the root node. Each node contains a header block and an array of variable length keys. The node header block contains a flag indicating whether the node is a leaf or not, a pointer to the node that is a parent to the current one, and a pointer to the node at the next lower level that contains keys that are less than the keys in this node. Each key in the array contains the string value of the key and a pointer to the node at the next lower level in the tree that contains keys greater than the current key and less than the next adjacent key in this array.

When the node is a leaf, the key pointers and the node header's pointer

to lower keys serve a different purpose. They contain some value that is relevant to the key. In some applications they will point to a data record. In others they will be the data record. Perhaps they point to a list of data records. The function of the key value is independent of the operation of the HyperTree. When you add a key, you add its associated key value. When you retrieve a key, you retrieve its value. The HyperTree, therefore, serves two retrieval objectives: It tells you if a key argument exists in the index, and it delivers the key value associated with key arguments.

Listing One, page 154, is <hyprtree.h>, the header file that describes the HyperTree structures. It defines several global values that you will change to suit your needs. The first one to consider is NODELEN. This is the byte length of a HyperTree node. If the node is too short, it will contain a small number of keys, and the tree will have many levels. This circumstance would affect retrieval performance because each search must navigate all the levels of the tree. If the node length is too long, the individual node searches, which are serial, would be affected.

The next value in <hyprtree.h> is MAXLEVELS, the maximum number of levels in a tree. This value is a function of the average number of keys in a node and the total number of keys in the index. You can conservatively set it to a safe high value such as 10. Ten levels of HyperTree would be a big index, indeed. The value itself is used as the dimension for an array of pointers. We need some value to use, because we do not know the upper limit until we reach it at run time.

MAXKEYLENGTH is used to limit the length of a key string. This value will depend on your application. It prevents a key from completely occupying a node. You should set it so that each node can contain at least three maximum-length keys. This will assure the correct behavior of the HyperTree algorithms.

You will need to consider the KEYVALUE *typedef* in <hyprtree.h>. As shown in the listing, KEYVALUE is a long integer. That usage will support the examples that follow, which use the KEYVALUE as a pointer into a file. KEYVALUE could be any valid C type including a structure. Remember that each leaf node will contain KEYVALUEs for the keys, so do not make them too long. If each key in your HyperTree is a vector to some complex data structure, put that structure into its own file and use the KEYVALUE as a pointer into that file.

Building a HyperTree

Listing Two, page 154, is <addkey.c>, the code needed to build a HyperTree. To prepare for this process, you need to have selected your keywords and phrases from the text database and sorted them into the collating sequence of the tree. The search algorithm described later assumes a case-insensitive collating sequence. You must provide the data value to associate with each keyword. That data type is described in <hyprtree.h> as the *typedef* KEYVALUE. The value will be the value returned by the searching algorithms. It could be a pointer to the text position in a text file, or it could be a pointer to a data structure that describes files, chapters, and paragraphs. Depending on the architecture of your Hypertext database, the KEYVALUE value could have the database document, chapter, and paragraph identifications encoded into its bit string. The important thing here is that you have prepared to build the HyperTree by extracting keywords, combining each of them with a data value to be returned, and sorting them. We'll have an example of that process later in this column.

The program searches the HyperTree by using the findkey *function*

In some applications, it might be appropriate to further process the keys following the sort. The duplicate values would be deleted, and the data value associated with each key value would become a pointer into a more complex data record or structure that describes the location and significance of the key value.

Building the HyperTree consists of reading the keys in sequence and filling nodes. When a node is full, the next value is added, in sequence, to the next higher node with the node number of the current node as its associated data value. Subsequent keys are added to a new node at the same level as the one that filled up. This upward growth is a recursive operation.

You call the *addkey* function to add nodes to a HyperTree. Its parameters are a pointer to the key's string and the KEYVALUE associated with the key. The first time you call this function, it creates the index file. When you are done adding keys, you call it with a NULL key string pointer to tell it to finish up. The *addkey* function calls the recursive *addkeyentry* function. This function is the tree-level manager for adding keys. The calls to it from *addkey* specify that keys are to be added to level zero. The *addkeyentry* function calls itself by specifying progressively higher levels as nodes fill and the tree grows.

Each node contains the node number of its parent. This data element supports the search of a HyperTree and so must be built when the tree is built. Because parents are growing when the children are fully grown — these metaphors are backwards, it seems — the program does not know the parent node number until the parent is filled. The *adopt* function takes care of that. When the *writenode* function writes a node, it scans the node and extracts the node numbers of each of its children. Then it calls the *adopt* function to write the parent's node number into each of the node records of the children.

Searching a HyperTree

Listing Three, page 156, is <srchtree.c>, which contains the functions that search a HyperTree. Several of the functions modify the position of the search pointers to a key. Others return the key's string or associated value.

The first search function is *findkey*. You pass it a pointer to a string that contains the key value you want to find. If it finds a match, it returns a true value. Otherwise it returns a false value. In either case, it positions the search pointers to the key that terminated the search. You may now call *current_value* to retrieve the KEYVALUE of the key that terminated the search, or *current_keystring* to retrieve the key's string value. This latter function is useful when *findkey* returns a false value to 4 say that it found no matching entry in the HyperTree. The terminating key is the next highest one in the collating sequence of the HyperTree.

There are other search functions that are used to navigate the HyperTree in its collated sequence. These are *firstkey*, *lastkey*, *nextkey*, and *prevkey*. They modify the search pointer's position, and subsequent calls to *current_ keyvalue* and *current_keystring* will reflect the new position. Each function returns a true value if it can satisfy the request. If *firstkey* or *lastkey* returns a false value, the index is empty. If *nextkey* returns a false value, the search pointers are already positioned at the end of the HyperTree's collating sequence. If *prevkey* returns a false value, the search pointers are already positioned at the beginning of the HyperTree's collating sequence.

The *findkey* search of a HyperTree

(continued from page 136)
begins at the root node. The search functions find the root node number by reading the HyperTree header record. To search the tree, the function searches the root node for a match on the argument. Keys are recorded in sequence in a node, so the search proceeds from left to right. If the key is not there, the function picks up the node pointer to the left of the key that terminated the search, reads that node, and starts over. This continues until the search either finds a match or terminates in a leaf node.

When you request the KEYVALUE by calling the *current_keyvalue* function, the search algorithm must navigate from the key's position in the HyperTree's system of levels to the leaf level. KEYVALUEs exist only at the leaf level. Higher levels contain pointers to lower levels. If you land on a key in a non-leaf node, you find the KEYVALUE by navigating down to the leaf this way. Begin by reading the node pointed to by the pointer just to the right of the matching key. Then read the node pointed to by the lower-key node pointer in the header block of the current node. Continue doing that until you reach a leaf. The lower-key pointer field contains (in a union) the KEYVALUE of the matching key. It doesn't make sense when I explain it, but it really does work that way.

A HyperTree Example

To put all this to use, I devised a simple hypertext-like application that builds a HyperTree index from a text file and lets you search it. We begin with some text file to test with. It should be made of unadorned ASCII with CR/LF pairs terminating each line. Use your editor to mark some key index values in the text by surrounding the keys with angle brackets (less-than, greater-than pairs). Make lots of keys. Now we can build a HyperTree.

Listing Four, page 158, and Listing Five, page 158, are <keyextr.c> and <keybuild.c>, two simple filter programs to build the HyperTree. You must link the object file from <keybuild.c> with that from <addkey.c>. Compile <keyextr.c> by itself. Run the two programs along with the SORT filter (assuming MS-DOS) this way:

```
keyextr < textfile.dat | sort | keybuild
```

where <textfile.dat> is the name of the file of ASCII text. The <keyextr> program extracts the key values you marked with angle brackets into a quoted string, which is followed by a comma and a numeric value that represents the key's position in the text file. The extracted

file would look like this:

```
"fortran", 125 "the merry widow",
257 "godfrey daniel", 3244
```

The SORT filter sorts the extracted values and pipes them to the <keybuild> program, which builds the HyperTree. That's all there is to that. You can look at the index with a dump utility. Its name is <hyprtree.ndx>.

To use the new HyperTree index, you will build the example program in Listing Six, page 159, <hyprsrch.c>. Its command line specifies the text file name and a key string to search, such as this one:

```
hyprsrch textfile.dat "windows on
                       the world"
```

The program searches the HyperTree by using the *findkey* function. It then displays a screen of text starting at the text position represented by the KEYVALUE of the key that terminated the search. The top of the screen shows the actual key value that was found along with the position. The bottom of the screen is a menu that lets you press F, L, N, P, or Q to move to the first, last, next, or previous key, or to quit and terminate the program.

HyperTree Extended

The small example we used here is merely a taste of how you can use the HyperTree data structure. The technique could be used as the retrieval engine for a spell checker or thesaurus program. It could be used as an inverted index to other documents from within a document. It could even be used in the concordance-like retrieval system that we built into TEXTSRCH.

There are some areas where HyperTree could be enhanced. Here are a few that come to mind.

In the example application, we do not bind the index to the file it indexes, requiring instead that you name the file on the command line. Your application would no doubt manage this association for the user. One way would be to include the text file's name as a part of the TREEHDR structure.

Because the HyperTree node contains keys in a collated sequence, you could add the data compression technique called "run length encoding" to the node structure. Many adjacent keys will begin with identical character sequences as in this list:

```
program
programmer
programming
progress
prototype
```

A simple escape sequence could specify how many characters the key inherits from the one that preceeds it. The list might then look this way:

```
program
\ 7mer
\ 8ing
\ 5ess
\ 3totype
```

The random-then-sequential properties of HyperTree make it a natural for wild card retrievals. If you use the ? character in the first position of a key ("?obbs," for example), you would do searches on all combinations of the key with letters and numbers in the first position. If the question mark was further into the key (for example, "Doc?or Dobbs"), you would use the *findkey* function to position yourself near the likely match and the *nextkey* function to retrieve all the possibilities, discarding those that do not fit.

Hypertext Editing

There are two programmer's editor tools — BTAGS and 4C — that offer Hypertext-like features to the PC programmer who is working with a system that involves a lot of source code files. We'll look at each of them in turn.

BTAGS is a program that you use along with the Brief programmer's editor. Brief is one of the most popular programmer's editors for the PC and it includes a comprehensive C-like macro language. BTAGS is a source file preprocessor and a set of Brief macros that implement a feature similar to the one used in the Unix vi editor. Here is how it works.

The BTAGS program builds a "tags" file by reading all your source files and finding where all the functions are defined. The tags file conforms to the Unix ctags format, which identifies each function identifier, the file where it is declared, and a string taken from the declaration. The string is built so that if the editor searches the file for the string, it will find the function declaration.

The BTAGS brief macros add command keys to the Brief editor. With the cursor on a function call, you press Ctrl-T, and the macro opens a new Brief window, reads the file where the function is declared, and positions the cursor on the function declaration. If you are not near a call to the function you want, Ctrl-E lets you type in a function name. Ctrl-B returns you to where you were before the last Ctrl-T or Ctrl-E.

BTAGS works only with function declarations and typedefs. It would be better if it also recorded external variables

and structure members. Another minor annoyance is that the Ctrl-T macro works only if the cursor is on the first character of a function name. You get the Brief macro language source code for the macros, so you could hack a fix to that one if you wanted to.

4C is a programmer's editor that is similar to BTAGS in that its source code analyzer program produces a ctags-compatible tags file. 4C produces ctags records for all the C language constructs, not just function declarations. 4C includes its own editor, which uses the tags file to allow you to jump around through your source files by putting the cursor anywhere on a variable name and pressing a function key. If you do prefer to use your own editor, you can tell 4C to call it instead of its own.

The 4C editor and its invocation of a user-specified editor employ a display strategy that I find unnatural. If you call up the main function, that's all you get. You cannot page beyond or ahead of it. If you call another function or variable, that's all you see. I find it frustrating to have to call a function by name when I already know that it appears within scrolling reach of where I am at the time.

At SD90 last February, the 4C folks were showing a new version of 4C that they plan on releasing later this year. The enhanced version provides interfaces to a variety of editors, including Brief, Multi-edit, Vedit, ME, and Slick, while the source-code analyzer supports C++, object-oriented Pascals, Modula-2, and ASM. The database query allows lookups by filename and directory and, for object-oriented languages, lookups by class and class hierarchy.

The upshot is that I like the BTAGS integration with Brief because Brief is the editor I use. I like 4C's inclusion of all the C constructs in their tags file. Both products use source analyzers that produce ctags-compatible tags files, so I tried the obvious. I used 4C to build a tags file and BTAGS brief macros to process them. Of course it did not work, but on the surface it looks like some hacking would bring it into line. Perhaps later.

Product Information

BTAGS
SD Enterprises
P.O. Box 621
Carpinteria, CA 93013
805-566-1317
Price: $49.95

4C
Tri-Technology Systems Inc.
1225 S. Elgin
Forest Park, IL 60130
312-366-7595
Price: $119

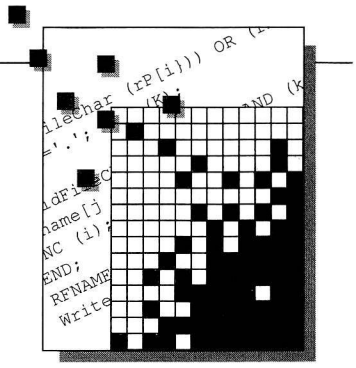
Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 154.)

Vote for your favorite feature/article.
Circle Reader Service **No. 10.**



Chasing Bubbles in the Waterbed

Jeff Duntemann, KI6RA/7

Cold fusion, huh? Well, I can top that: I have a waterbed that breaks down water into its component gases without electricity. I'm not sure what these gases are. They could be hydrogen and oxygen, or I suppose they could as well be carbon dioxide and xenon. I'm not macho enough to perform any conclusive experiments. About all I am sure of is that this particular waterbed has generated what seems like hundreds of cubic feet of invisible gas since being filled with equal parts water and Algae-Go-Bye-Bye in May of 1987.

Every week or so I have to chase Mr. Byte off the comforter, yank the bedclothes off the waterbed, and wheedle ten zillion little bubbles around under the vinyl with my old slide rule, merging tiny bubbles into small bubbles, small bubbles into larger bubbles, larger bubbles into Godzillan bubbles, and finally letting the single monstrous survivor out through the fill hole. Otherwise, an ordinary night between the sheets can sound like one of those old Sea Hunt episodes.

Dynamic Variables Recap

Ever alert for the occasional wild metaphor roosting in the rafters, it occurred to me that we have this same problem in the structured programming world. The faulty waterbed is the heap, and the bubbles — like the 'ole that Ringo 'ad in his pocket in Yellow Submarine — are simply blocks of empty space that have been used for awhile and then turned loose.

First, a little background for those who work in languages such as Basic that do not support explicit memory allocation and deallocation. Pascal and Modula-2 (and C as well) set aside a certain amount of memory called the "heap" and allow the running program to create variables out of that memory (we say "on the heap") as needed. These variables are called dynamic variables to differentiate them from static variables, which are named in the source code and allocated at compile time, and exist in the same form and in the same location as long as the program runs. When no longer needed, dynamic variables are destroyed, or deallocated, and the memory they occupied on the heap is made available as grist from which to create other variables. In both Pascal and Modula-2 the allocation routine is called *New*, and the deallocation routine is called *Dispose*. (Turbo and QuickPascal have a similar pair of routines called *GetMem* and *FreeMem* as well.)

Variables created on the heap have no names. They are accessed through a type of variable called a "pointer" that exists solely to "tie" these dynamic variables into the program's reality. The pointer (which is declared in the usual way variables are declared) is set to point to the dynamic variable when the dynamic variable is allocated using *New*:

```
VAR
   { A pointer to type Integer: }
   MyPointer : ^Integer;
.
.
.
New(MyPointer);
MyPointer^ : = 42
```

The caret symbol is the dereference operator. To specify what the pointer points to (rather than to specify the pointer itself) you must use the dereference operator, as shown in the assignment statement. Accessing the item that a pointer points to is called "dereferencing the pointer" and the item pointed to is called the "pointer's referent."

If you could only create dynamic variables as referents of pointers allocated as static variables, it would be interesting but not compellingly useful. Pointers really shine when they are used to connect the components of entire data structures constructed on the heap. Linked lists, queues, stacks, and other data structures can be built from simple data structures and pointers. This is done by making pointers the fields of records:

```
TYPE
   DynaPtr = ^DynaRec;

   DynaRec =
      RECORD
         StrData : STRING;
         Next : DynaPtr;
      END
```

In a linked list, several of these records are allocated on the heap such that the *Next* pointer in each record points to the "next" record in the list, allocated elsewhere on the heap. The pointer in the last record is set to a value of *NIL*, which is a sentinel value indicating that the pointer points to nothing.

Standard procedures exist to query the system and find out how much free heap memory is available. The *MemAvail* function returns the total number of bytes of free memory on the heap. The *MaxAvail* function returns the size in bytes of the largest single chunk of memory on the heap.

Heaps of Holes

This is a powerful concept, but there's a worm in it: When you deallocate items stored on the heap, the memory where the item had been, becomes a "hole," the exact size of the item. This hole is available for use in allocating future dynamic variables, but obviously, you can't allocate a variable any larger than

the variable that created the hole by going poof.

This sounds worse than it often is. Items are allocated on the heap in order, side by side. If you allocate 100 records in a row, use them, and then deallocate all 100 records at once; the "hole" resulting from the deallocation will be contiguous, the size of the full 100 records. Problems arise when you allocate a great many items of varying sizes on the heap, and deallocate some but not others, especially if the deallocation is fairly random. Allocating a record in a hole left by a slightly larger record can leave a tiny sliver of unused memory that isn't good for anything at all. Do this often enough, and you may have a great deal of free memory that can't be used at all because it exists in a large number of very small chunks.

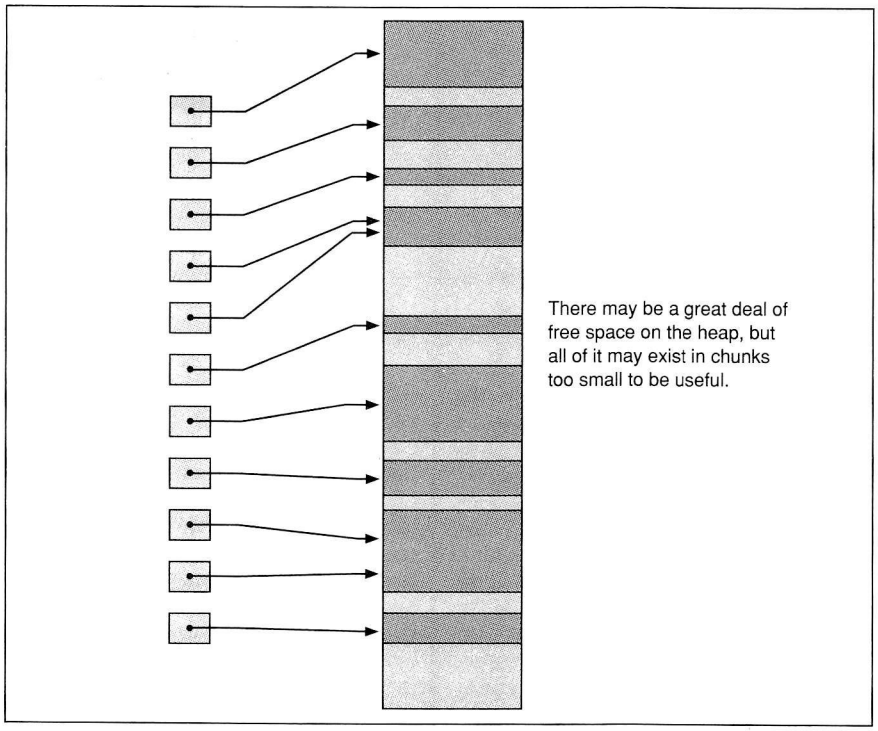


Figure 1: *A fragmented heap*

This condition is called "heap fragmentation" because the heap becomes divided into a great many separate fragments of memory in varyingly useful sizes. Figure 1 represents this condition in Pascal or Modula-2. The shaded areas represent memory that is in use, pointed to by one of the pointers on the left. The white areas represent memory that is free for use. Note that some of the slices are pretty thin, and may not be large enough to hold anything useful to the currently running application.

The Long Wait for the Garbage Man

So, what happens if you call *MaxAvail* and the size of the largest free chunk of heap memory is less than the size of the item you need to allocate on the heap? Not much. The best you can do is to try deallocating some other items on the heap and hope that they'll open up a hole large enough to fit the item you need to create. If there's nothing you can turn loose, you're stuck. Really.

When confronted with this problem, most people assume there must be a way to rearrange the blocks of memory on the heap, packing them up nose-to-tail so that all the little empty slivers of memory collect in one place and add up to a single large block of useful space again. It's pretty disconcerting to realize that there is absolutely no general-purpose way to do this in Pascal or Modula-2. (Or C, either.) Every trick involves making assumptions about the order in which items were allocated on the heap and other information that varies from application to application or even from one run time to the next.

This Holy Grail of gathering together all the bubbles in the great waterbed of heap memory is what we call "garbage collection." It is one of the most wretchedly difficult things to do in all computer science, and in many languages it is simply impossible.

Why? It has to do with the way pointers are implemented in our most common implementations of Pascal and Modula-2. Pointers are simply machine addresses. The "long" pointers used in Modula-2 (and all pointers in Turbo

(continued from page 144)
and QuickPascal) are 32-bit addresses consisting of a 16-bit segment address and a 16-bit offset address. So a pointer is just 4 bytes somewhere in the megabyte of 8086 real address space, containing the address of the memory block that the pointer points to. (A *NIL* pointer contains 4 bytes of 0s.)

The problem with rearranging the heap lies in notifying the pointers that the addresses of their referents have changed. A pointer *is* an address, so you would have to change the value of every pointer that pointed to any block of heap memory involved in a move. The problem with that is simply finding all the pointers. Although you can always find a pointer's referent starting from the pointer, you can't work back from a block of memory to find all pointers that point to it. Keep in mind that any number of pointers may point to a single block of memory on the heap, and those pointers may be anywhere in memory at all, including on the heap, in the code segment, or on the stack. Nothing marks a pointer as being a pointer; furthermore, nearly all 4-byte sequences in memory contain binary patterns that could represent valid addresses.

So there may be tens of thousands of pointers scattered through the megabyte of real address space, and they look no different from 4 bytes of machine code, 4 bytes of data, or 4 bytes of anything. Finding pointers just by looking for them is thus meaningless. And if you can't find every pointer that points to a block of data, you had better leave that block of data right where it is. That in a nutshell is why garbage collection for the standard Pascal/Modula-2 heap is impossible.

What Those Other Guys Do

Outside of the Pascal/C/Modula world things are considerably better. Smalltalk and Actor both support fully general automatic garbage collection on their dynamic storage. One reason both pick up their trash is that they have to; both are large, ambitious systems that take up a lot of memory. Without garbage collection, they would drink their heap dry in no time flat.

Actor contains a mechanism that constantly scans its stack and dictionary, looking for "dead" objects to which no references are found. Actor's dictionary is roughly equivalent to a symbol table. If a block of memory is allocated for an object, and that object is not referenced in the dictionary or by any other object, the block is scavenged and made available for new objects.

This happens automatically, and the process continues in the background throughout the execution of an Actor program. This way, the garbage collection overhead (which takes an irritatingly large fraction of the CPU cycles devoted to a program's execution) is spread out evenly through a program's life and is thus less noticeable. Some older systems performed garbage collection all at once, which caused the executing program to stop dead in its tracks (sometimes for minutes at a time) while the garbage collector fiddled memory around. This happened to me once back in my Xerox days while I was learning some internal revision of Smalltalk on a creaky old Alto workstation. I was certain the system had croaked, but lo! It was only picking up its dirty socks, and came back to me after what had to be six or seven minutes.

Actor's garbage collector works so well that there is no dispose message to clean up after allocations triggered by *New*; when a program no longer needs to work with an object, it cuts the object's space loose by setting its internal state to *Nil*. The scavenger then picks up the trash on its next pass.

Smalltalk/V's documentation says less about its garbage collection system than Actor's, but things seem to work in about the same way: When an object is no longer referenced, it is removed from memory and the memory is reclaimed. Unused symbols do accumulate, however, and must be explicitly purged and their space reclaimed by sending the *purgeUnusedSymbols* message to the Smalltalk system.

The Secret of the Middleman

There are lots of ways to implement garbage collection, most of them far beyond my understanding. What I will explain, though, is the minimum machinery that makes garbage collection possible. What you need, basically, is a middleman.

I've drawn such a middleman in Figure 2. Between the pointers scattered through the system and the heap itself is an array of handles. I've shown the handles as pointers themselves, but they aren't pointers in the same physical sense that Pascal and Modula-2 pointers are. A handle is means of access. It could be a pointer to the actual block of heap memory, or it could be a pointer into some larger mechanism that maps blocks of storage onto disk sectors, or EMS or extended memory. Also, a handle must contain the size of the memory block it points to. I've not shown this in the figure for simplicity's sake, but a handle might be a very simple record containing a pointer to the memory block on the heap and a 16-bit block size value.

A handle's primary virtues are two: First of all, for each block of memory there is only *one* handle, and second, the system always knows where every handle is. I show an array of handles in the figure for simplicity; in many cases the handles will be arranged as

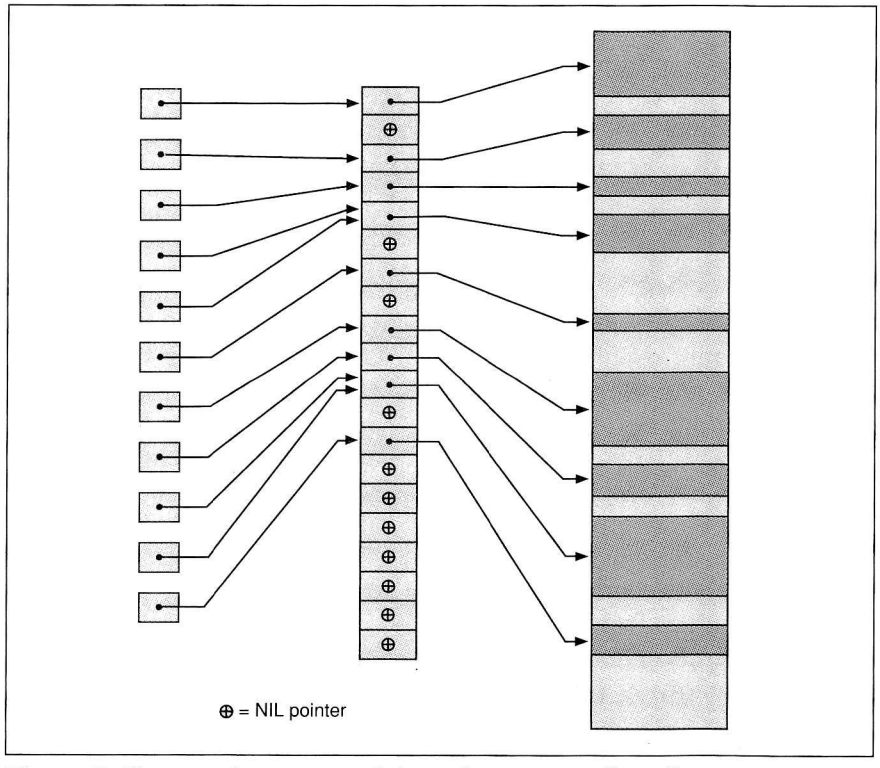


Figure 2: *The same heap accessed through an array of handles*

a linked list. The important thing is that the handles are stored in a form that is always under the system's control.

A handle either points to a block of memory on the heap or it is set to *NIL* to indicate that the handle is free and may be used. Note in Figure 2 that in two cases, two different pointers point to the same handle. This is entirely equivalent to the situation in Figure 1 where two pointers pointed to the same block of memory on the heap. Any number of pointers may point to one handle, but only one handle may point to any given block of memory.

Packing the Heap

The heap shown in Figure 2 is identical to that shown in Figure 1, that is, fragmented to the point of being useless. Unlike Figure 1, however, the fragmentation in Figure 2 can be fixed by a little scavenging. Because the system knows where every handle is located, and because each handle points to one memory block and each memory block has only one handle pointing to it, the system can eliminate wasted space by moving memory blocks together on the heap until they become contiguous. This condition is shown in Figure 3. All of the available heap space is now in one large block, and no small slivers of memory remain wasted.

In a practical system, a few more things would be necessary, such as a "free list" to keep track of available blocks. Both Turbo Pascal and QuickPascal use such a list, a simple linked

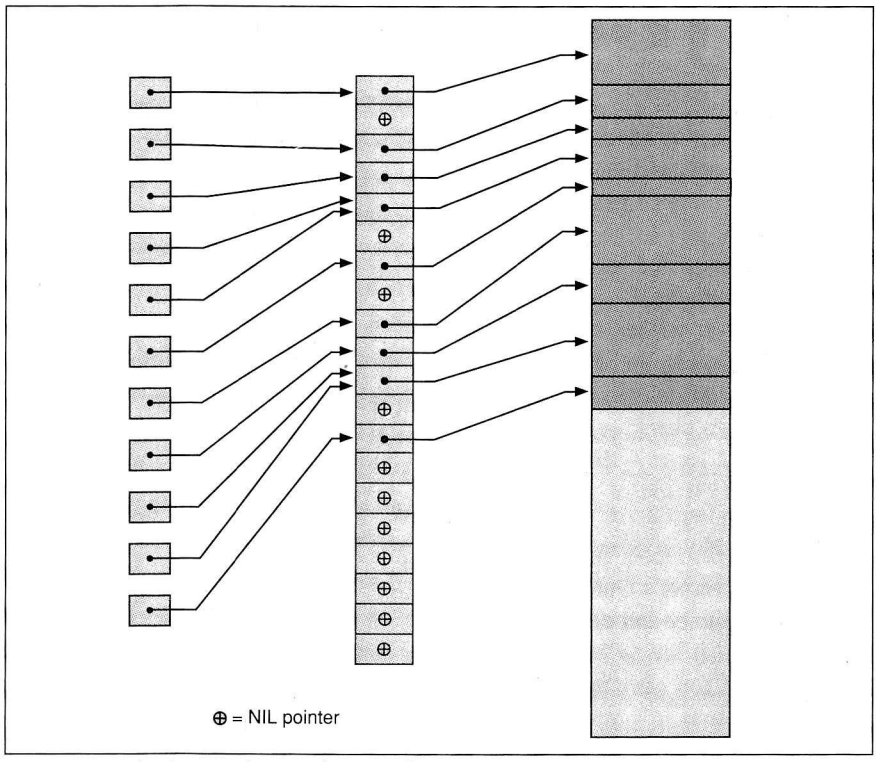


Figure 3: *The heap after garbage collection*

list of records located on the heap.

Could a system such as this be built into a Pascal or Modula-2 compiler? Of course. The cost is in speed, and, to a lesser extent, in the memory needed by the handles and the code to manipulate them. Dereferencing a pointer to a pointer involves two separate memory accesses instead of only one. Also, as mentioned before, the garbage collection task itself takes some time, but with some cleverness can be spread so thin as to hardly be noticeable, à la Actor.

Breaking Old Habits

The real problem, though, is breaking current code. Turbo Pascal, in particular, allows a lot of direct manipulation of pointers. If it were as simple as generating handle-manipulation code for *New*, *Dispose*, and the dereference operator, there'd be little difficulty with compatibility. However, a great deal of Turbo Pascal code (including a lot of my own) make assumptions about the nature of pointers that would not jive with a handle-based heap manager.

The most obvious case is building pointers from segment and offset addresses using the *Ptr* function. How many times have you done something like this:

```
VAR
   Display : Pointer;
   .
   .
   .
Display := Ptr($B800,0);
```

The last thing you want is for a heap manager to decide to move your video refresh buffer somewhere a little closer to the other bubbles.

The kicker is that pointers are often used for things that have nothing at all to do with the heap. And as long as pointers point to things that exist at fixed locations and cannot (or should not) be moved, using handles for heap management with traditional pointer and dereferencing syntax is going to be difficult indeed.

The Trouble with Objects

I'm making a very big deal of all this because sometime soon the matter of heap fragmentation is going to become a very serious problem for the newborn object-oriented languages. I won't speak for C++ because I don't understand it very well as yet (though I'm trying) but for Turbo and QuickPascal we're headed for trouble.

Heap fragmentation has been with us from the beginning, but it's attracted little attention for two major reasons:

1. Many self-taught Pascal programmers hit a conceptual wall when they encounter pointers, and rather than figure them out or yell for help, simply work around them. Thus, a great many Pascal applications don't make use of the heap at all.

2. Seasoned Pascal developers who use the heap heavily have worked out tricks to deal with heap fragmentation. These include padding records out so that most records allocated on the heap are either the same size or convenient multiples of the size of the smallest record, or massaging an algorithm such that records are allocated and deallocated in order rather than at random. "Heap discipline" of this sort is second nature to longtime Pascal developers, and is considered by many to be part of good structured programming practice, even though it violates the spirit of true dynamic allocation.

Unfortunately, when you start dealing with objects in a big way, this kind of heap discipline no longer works. A linked list of objects no longer contains items all of one type. As long as all objects in a list are descended from a common ancestor (such as the *Node* type shipped as an example with Turbo Pascal 5.5) polymorphism allows the developer to stop worrying about the type of the nodes in a list and let the nodes handle their own business

through virtual methods. When the application has to have carnal knowledge of the contents of a list in order to make things happen, most of the benefits of object-oriented techniques get lost. Much of OOP's novelty lies in giving individual objects more autonomy, but heap discipline generally means orchestrating heap management in ways that individual objects — being just parts of a larger whole — cannot accomplish.

Compounding the problem is the fact that programmers can no longer ignore the heap once they start using OOP. In QuickPascal *all* objects are allocated on the heap. You don't get any choices. In Turbo Pascal, objects may be defined statically in the data segment without involving the heap if you like, but such objects may not take part in polymorphic algorithms and are considerably less useful than objects on the heap.

So picture it: An ambitious application consisting of dozens or hundreds of objects popping into being on the heap or poofing into holes more or less at random, without the application's being fully aware of individual objects' exact types and hence their sizes. The old tricks no longer work, and (to state it in line with the metaphor of active data) the heap fragments itself to uselessness in record time.

The Tyranny of the Installed Base

Both Turbo Pascal 5.5 and QuickPascal suffer from this problem, but if a solution is out there, QuickPascal will be the easier fix by far. Because the installed base for QuickPascal is counted in the tens of thousands rather than in the many hundreds of thousands, there is less QuickPascal code to break and fewer QuickPascal users to alienate if a fundamental syntactic change must be made to the language to allow a handle-based heap manager.

But — most remarkably — QuickPascal may not need to make any syntactic changes at all. What I originally thought was oversight or sheer clumsiness in the Apple Pascal definition (which QuickPascal follows closely) might hold the solution to the whole mess.

Recall the inconsistent nature of object definition in QuickPascal. Objects are defined like records, allocated like pointers, and then used like records:

```
TYPE
  Figure = OBJECT
             X,Y : Integer;
             Visible : Boolean;
             PROCEDURE Show;
             PROCEDURE Hide;
           END;

VAR
  MyObject = Figure;
  .
  .
  .
New(MyObject);
MyObject.A := 17;
MyObject.Y := 42;
MyObject.Show;
```

Ordinarily, *New* works only with pointers, but QuickPascal allows *New* to take an object variable identifier as though it were a pointer. In fact, beneath the surface the variable *MyObject* is a pointer, but once passed to *New* it can be used as though it were the object itself and not simply a pointer to a nameless block of memory on the heap.

What's important is that, regardless of its physical implementation, a QuickPascal object is not accessed through pointers. Unless you explicitly declare an object as the referent of a pointer type, the dereference operator is not required to access the object's methods and fields. This means that, if Microsoft chose to do so, it could isolate objects in an objects-only heap, and implement a handle-based manager for objects that could include automatic garbage collection. The name of the object would become a pointer to a handle, which would then point to the object's actual location on the heap. This does not involve redefining pointer syntax or the dereference operator, and would not break a single line of Version 1.0 code. For QuickPascal it would solve the heartbreak of heap fragmentation for objects, which is itchier than psoriasis and lots harder to get rid of.

I won't say that this was what the designers of Apple Pascal had in mind when they chose their somewhat idiosyncratic syntax for object creation (though if they contact me I'd love to ask them) but it certainly is a potential solution aching to be implemented. It will have some costs in performance, but I suppose we were naive in assuming that the amazing flexibility of polymorphism would come for free.

Turbo Pascal's designers will have a much harder row to hoe. There is no solution that won't involve breaking megalines of existing code. They're clever, those Borlanders; as clever as you find in this business and then some, but the tyranny of the installed base is going to make the Big Fragmentation Fix for Turbo Pascal a painful one all the way around.

Would that it were as simple as chasing bubbles in a waterbed.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 11.**

Listing One *(Text begins on page 153.)*

```
on mousewithin
  --
  --hypertext technique by Steve Drazga, AnalytX
  --if you use this in your scripts please include these 2 lines.
  --
    if the locktext of the target is true then
      set locktext of target to false --unlock the field if it is
locked
   end if

   if selection is not empty then --something was selected
      put selection into SelectedWord
      if space is in SelectedWord then --user selected > 1 word
         click at loc of target --so we will clear the selection
         exit mousewithin --and exit to wait for another
selection
      end if
    --
    --this is the section where you do something with the selection
    --You can bring up a pop up note or you can go to another card.
    --
    end if
end mousewithin
```

End Listing One

Listing Two

```
on mouseUp

  -- This code, placed in a button script, implements
  -- Harvey Chang's hypertext trick.  The user selects
  -- any text in a field and clicks on the button.
  -- The script first tries to use the selected text as a
  -- hypertext link, then falls back to simple search.

  doMenu Copy Text
  put "Montreal Hypertext, Harvey Y Chang MD, 1988 Jan 16"
  push card
  go to Montreal Hypertext Demo
  doMenu Find...
  doMenu Paste Text
  put " in field " & quote & "Title" & quote after message
  do message
  if the result is "not found" then
    answer "not found in Titles:  search text?" with "OK" or "No"
    if it is "No" then
      pop card
      exit mouseUp
    else
      doMenu Find...
      doMenu Paste Text
      put " in field " & quote & "Text" & quote after message
      do message
    end if
  end if
end mouseUp
```

End Listing Two

Listing Three

```
on mouseUp

  -- This is a scrolling field script.  Its field must be locked.
  -- It implements an index field, to be placed on the first
  -- card of the stack to be indexed.  This is the index card.
  -- When the mouse is clicked inside the field, this script causes
  -- a jump to the card corresponding to the line clicked on.
  -- The line commented out uses the text in the line,
  -- rather than its number, as the link.

  go to card getLineNum(the mouseV)
  -- find line getLineNum(the mouseV) of me in field keyword

end mouseUp

function getLineNum mouseVert

  -- Returns the number of the line clicked on.

  -- It works like this:
  -- Subtracting the top of the field and its scroll from
  -- the mouse's vertical location gives the
  -- mouse's vertical location within the field.
  -- Dividing this by the textHeight of the field & adding 0.5
  -- converts pixel counts to line counts.
  -- Rounding gives a value acceptable as a card number.

  -- Note: although this technique should work with any font size,
  -- turning on WideMargins will confuse the count.
  -- To adapt this script to a non-scrolling field,
  -- remove "+ the scroll of me" from the computation.

  return round(((mouseVert - the top of me + the scroll of me) /
(the textHeight of me)) + 0.5)

end getLineNum
```

End Listings

Listing One *(Text begins on page 135.)*

```
/* ---------- hyprtree.h --------- */

#define NODELEN 256         /* length of a node   */
#define MAXLEVELS 10        /* tree levels        */
#define MAXKEYLENGTH 25     /* maximum key length */

#define TRUE 1
#define FALSE !TRUE

#define KSPACE (NODELEN-sizeof(NODEHDR))
#define HYPERTREE "hyprtree.ndx"

/* ----- computes length of a key in a node -------- */
#define keylength(nd,kp) (strlen(kp) + 1 + \
    ((nd).nodehdr.isleaf?sizeof(KEYVALUE):sizeof(NODEPOINTER)))

/* ---------- node pointers and key values ---------- */
typedef long KEYVALUE;
typedef int NODEPOINTER;

typedef union {
    KEYVALUE keyvalue;
    NODEPOINTER nodepointer;
} KEYPOINTER;

/* ---------- header record for a hypertree ---------- */
typedef struct {
    NODEPOINTER rootnode;
} TREEHDR;

/* ---- header structure for a hypertree node -------- */
typedef struct  {
    int isleaf;         /* true if the node is a leaf */
    NODEPOINTER parent; /* node number of parent */
    KEYPOINTER lower;   /* keys < this node (or value if leaf)*/
} NODEHDR;

/* --------- node record for a hypertree ---------- */
typedef struct {
    NODEHDR nodehdr;
    char keys[KSPACE];
} TREENODE;

/* ------------- prototypes --------------- */
void addkey(char *key, KEYVALUE keyvalue);
int findkey(char *key);
int firstkey(void);
int lastkey(void);
int nextkey(void);
int prevkey(void);
KEYVALUE current_keyvalue(void);
char *current_keystring(void);
```

End Listing One

Listing Two

```
/* ---------------- addkey.c -------------- */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hyprtree.h"

static void addkeyentry(int level, char *key,
            KEYPOINTER keypointer, NODEPOINTER lowernode);
static void nullnode(int level);
static int freespace(TREENODE node);
static void writenode(int level);
static void adopt(NODEPOINTER child, NODEPOINTER parent);

static TREENODE *nodes[MAXLEVELS];
static NODEPOINTER nodenbr[MAXLEVELS];
static NODEPOINTER nextnode;

static FILE *htree;
static TREEHDR th;

/*
 *  Add a key string value and its associated KEYVALUE to
 * the hypertree.
 */
void addkey(char *key, KEYVALUE keyvalue)
{
    KEYPOINTER keypointer;
    if (key == NULL)    {
        /* ------ terminal key, complete the tree ------ */
        if (htree != NULL)  {
            int level = 0;
            /* -------- write the unwritten nodes -------- */
            while (nodes[level] != NULL)    {
                writenode(level);
                free(nodes[level]);
                level++;
            }
            /* ------ update the header block ---------- */
            th.rootnode = nodenbr[level-1];
            fseek(htree, 0L, SEEK_SET);
            fwrite(&th, sizeof(TREEHDR), 1, htree);

            fclose(htree);
        }
    }
    else    {
        keypointer.keyvalue = keyvalue;
        if (strlen(key) <= MAXKEYLENGTH)
            addkeyentry(0, key, keypointer, 0);
    }
}

static void addkeyentry(int level, char *key,
                KEYPOINTER keypointer, NODEPOINTER lowernode)
{
    char *kp;

    if (nodes[level] == NULL)   {
        /* -------- build a new node ---------- */
        if (level == 0) {
            /* ------ building a new tree ------ */
            htree = fopen(HYPERTREE, "wb+");
            /* ----- write a NULL header for now ----- */
            fwrite(&th, sizeof(TREEHDR), 1, htree);
        }
        if ((nodes[level] = malloc(sizeof(TREENODE))) == NULL)  {
            fputs("\nOut of memory!", stderr);
            exit(1);
        }
        nodes[level+1] = NULL;
        nullnode(level);
        /* ---- point to the node of the lower keys --- */
        nodes[level]->nodehdr.lower.nodepointer = lowernode;
        /* --- assign the next node number to this node --- */
        nodenbr[level] = ++nextnode;
    }
    if (keylength(*nodes[level],key) >
            freespace(*nodes[level]))   {
        /* -------- this node is full -------- */
        KEYPOINTER keyp;
        NODEPOINTER lowernode;
        /* ---- write the node to the index file ---- */
        writenode(level);
        /* ---- remember the node nbr at this level ---- */
        lowernode = nodenbr[level];
        /* --- assign a new node number to this level --- */
        nodenbr[level] = ++nextnode;
        /* --- grow or add to parent with the current key --- */
        memset(&keyp, 0, sizeof(KEYPOINTER));
        keyp.nodepointer = nodenbr[level];
        addkeyentry(level+1, key, keyp, lowernode);
        /* - now set the node at this level to NULL values - */
        nullnode(level);
        nodes[level]->nodehdr.lower = keypointer;
    }
    else    {
        /* ------ insert the key into the node ------ */
        kp = nodes[level]->keys;
        /* ----- scan to the end of the node ----- */
        while (*kp)
            kp += keylength(*nodes[level], kp);
        strcpy(kp, key);
        /* ----- attach the key pointer to the key ----- */
        kp += strlen(kp)+1;
        if (level == 0)
            *((KEYVALUE *)kp) = keypointer.keyvalue;
        else
            *((NODEPOINTER *)kp) = keypointer.nodepointer;
    }
}

/* --------- build a null node ------------ */
static void nullnode(int level)
{
    memset(nodes[level], 0, NODELEN);
    nodes[level]->nodehdr.isleaf = level == 0;
}

/* ------ compute space remaining in a node ------- */
static int freespace(TREENODE node)
{
    int sp = KSPACE;
    char *kp = node.keys;

    while (*kp) {
        sp -= keylength(node,kp);
        kp += keylength(node,kp);
    }
    return sp;
}

/* ------ write a completed node ------- */
static void writenode(int level)
{
    long where = (nodenbr[level]-1)*NODELEN+sizeof(TREEHDR);

    fseek(htree, where, SEEK_SET);
    fwrite(nodes[level], NODELEN, 1, htree);
    if (level)  {
        /* - this is a parent node, update its children - */
        char *kp = nodes[level]->keys;
        adopt(nodes[level]->nodehdr.lower.nodepointer,
                    nodenbr[level]);
        while (*kp) {
            adopt(*((NODEPOINTER *)(kp+strlen(kp)+1)),
                    nodenbr[level]);
            kp += keylength(*nodes[level], kp);
        }
    }
}

/* ---- write a parent node number into a child node ---- */
static void adopt(NODEPOINTER child, NODEPOINTER parent)
{
    NODEHDR nodehdr;
    long here = ftell(htree);
    long where = (child-1)*NODELEN+sizeof(TREEHDR);
```

(Listing continued on page 156)

Listing Two *(Listing continued, text begins on page 135.)*

```
    fseek(htree, where, SEEK_SET);
    fread(&nodehdr, sizeof(NODEHDR), 1, htree);
    nodehdr.parent = parent;
    fseek(htree, where, SEEK_SET);
    fwrite(&nodehdr, sizeof(NODEHDR), 1, htree);
    fseek(htree, here, SEEK_SET);
}
```

End Listing Two

Listing Three

```
/* ---------- srchtree.c ---------- */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hyprtree.h"

/* ---------- search packet ---------- */
typedef struct  {
    NODEPOINTER node;
    char *ky;
} SEARCH;

FILE *htree;
static TREEHDR th;
static TREENODE node;
static SEARCH current;

static void readnode(NODEPOINTER nodepointer);
static void opentree(void);

/*
 * Find a specified key in the tree.
 * Return TRUE if found.
 * Return FALSE if not found.
 */
int findkey(char *key)
{
    opentree();
    if (th.rootnode)    {
        SEARCH save = current;
        readnode(th.rootnode);
        while (TRUE)    {
            int cmp;
            current.ky = node.keys;
            while (*current.ky) {
                cmp = stricmp(key, current.ky);
                if (cmp < 0)    {
                    save = current;
                    break;
                }
                if (cmp == 0)
                    return TRUE;
                current.ky += keylength(node, current.ky);
            }
            if (!node.nodehdr.isleaf)   {
                if (current.ky == node.keys)
                    readnode(node.nodehdr.lower.nodepointer);
                else
                    readnode(*((NODEPOINTER *)
                        (current.ky-sizeof(NODEPOINTER))));
            }
            else    {
                current = save;
                readnode(current.node);
                break;
            }
        }
    }
    return FALSE;
}

/*
 * Find the first sequential key in the tree.
 * Return TRUE if found.
 * Return FALSE if not found (the tree is empty).
 */
int firstkey(void)
{
    opentree();
    if (th.rootnode)    {
        readnode(th.rootnode);
        while (!node.nodehdr.isleaf)
            readnode(node.nodehdr.lower.nodepointer);
        current.ky = node.keys;
        return TRUE;
    }
    return FALSE;
}

/*
 * Find the last sequential key in the tree.
 * Return TRUE if found.
 * Return FALSE if not found (the tree is empty).
 */
int lastkey(void)
{
    NODEPOINTER np;
    opentree();
    if (th.rootnode)    {
        int len = 0;
        np = th.rootnode;
        current.node = th.rootnode;
        current.ky = node.keys;
        do  {
```

```
            SEARCH save = current;
            readnode(np);
            current.ky = node.keys;
            while (*current.ky) {
                len = keylength(node, current.ky);
                current.ky += len;
            }
            if (current.ky == node.keys)    {
                readnode(save.node);
                current.ky = save.ky;
                break;
            }
            else
                np = *((NODEPOINTER *)
                    (current.ky-sizeof(NODEPOINTER)));
        } while (!node.nodehdr.isleaf);
        current.ky -= len;
        return TRUE;
    }
    return FALSE;
}

/*
 * Find the next sequential key in the tree.
 * Return TRUE if found.
 * Return FALSE if not found (the tree is empty or at the end).
 */
int nextkey(void)
{
    opentree();
    if (th.rootnode)    {
        if (current.ky == NULL)
            return firstkey();
        current.ky += keylength(node, current.ky);
        if (!node.nodehdr.isleaf)   {
            readnode(*((NODEPOINTER *)
                    (current.ky-sizeof(NODEPOINTER))));
            current.ky = node.keys;
            while (!node.nodehdr.isleaf)
                readnode(node.nodehdr.lower.nodepointer);
        }
        /* ----- while at the end of a node ------ */
        while (*current.ky == '\0') {
            NODEPOINTER child = current.node;
            if (node.nodehdr.parent == 0)
                break;
            readnode(node.nodehdr.parent);
            current.ky = node.keys;
            if (child == node.nodehdr.lower.nodepointer)
                break;
            while (*current.ky) {
                NODEPOINTER this = *((NODEPOINTER *)
                    (current.ky+strlen(current.ky)+1));
                current.ky += keylength(node, current.ky);
                if (this == child)
                    break;
            }
        }
        return *current.ky != '\0';
    }
    return FALSE;
}

/*
 * Find the previous sequential key in the tree.
 * Return TRUE if found.
 * Return FALSE if not found
 * (the tree is empty or at the beginning).
 */
int prevkey(void)
{
    char *kp;
    opentree();
    if (th.rootnode)    {
        if (current.ky == NULL)
            return lastkey();
        if (!node.nodehdr.isleaf)   {
            /* ----- navigate to end of the lower leaf ----- */
            NODEPOINTER np;
            if (current.ky == node.keys)
                np = node.nodehdr.lower.nodepointer;
            else
                np = *((NODEPOINTER *)
                    (current.ky-sizeof(NODEPOINTER)));
            while (!node.nodehdr.isleaf)    {
                readnode(np);
                current.ky = node.keys;
                while (*current.ky)
                    current.ky += keylength(node, current.ky);
                np = *((NODEPOINTER *)
                    (current.ky-sizeof(NODEPOINTER)));
            }
        }
        /* ----- while at the beginning of a node ------ */
        while (current.ky == node.keys) {
            NODEPOINTER child = current.node;
            if (node.nodehdr.parent == 0)
                break;
            readnode(node.nodehdr.parent);
            current.ky = node.keys;
            if (child == node.nodehdr.lower.nodepointer)
                continue;

            while (*current.ky) {
                NODEPOINTER this = *((NODEPOINTER *)
                    (current.ky+strlen(current.ky)+1));
                current.ky += keylength(node, current.ky);
                if (this == child)
                    break;
            }
```

(Listing continued on page 158)

Listing Three *(Listing continued, text begins on page 135.)*

```
        }
        /* -------- go to previous key in node -------- */
        if (current.ky != node.keys)    {
            kp = node.keys;
            while (kp+keylength(node, kp) < current.ky)
                kp += keylength(node, kp);
            current.ky = kp;
            return TRUE;
        }
    }
    return FALSE;
}

/*
 * Return the key value associated with the most recently
 * retrieved key.
 */
KEYVALUE current_keyvalue(void)
{
    KEYVALUE rtn;
    if (!node.nodehdr.isleaf)   {
        SEARCH save = current;
        current.ky += keylength(node, current.ky);
        while (!node.nodehdr.isleaf)    {
            NODEPOINTER np;
            if (current.ky == node.keys)
                np = node.nodehdr.lower.nodepointer;
            else
                np = *((NODEPOINTER *)
                    (current.ky-sizeof(NODEPOINTER)));
            readnode(np);
            current.ky = node.keys;
        }
        rtn = node.nodehdr.lower.keyvalue;
        current = save;
        readnode(save.node);
        return rtn;
    }
    return *((KEYVALUE *)(current.ky+strlen(current.ky)+1));
}

/*
 * Return the key string value of the most recently
 * retrieved key.
 */
char *current_keystring(void)
{
    return current.ky;
}

/* -------- open the tree ----------- */
static void opentree(void)
{
    if (htree == NULL)  {
        if ((htree = fopen(HYPERTREE, "rb")) == NULL)   {
            fputs("\n" HYPERTREE " not found", stderr);
            exit(1);
        }
        fread(&th, sizeof(TREEHDR), 1, htree);
    }
}

/* ---------- read a specified node ------------ */
static void readnode(NODEPOINTER nodepointer)
{
    long where = (nodepointer-1)*NODELEN+sizeof(TREEHDR);

    fseek(htree, where, SEEK_SET);
    fread(&node, NODELEN, 1, htree);
    current.node = nodepointer;
}
```

End Listing Three

Listing Four

```
/* -------- keyextr.c ---------- */

/*
 * Build a test HyperTree from standard input.
 * <> delimiters define the key values.
 * This is pass one, which builds the sortable table.
 */

#include <stdio.h>
#include <string.h>
#include "hyprtree.h"

#define KEYTOKEN '<'
#define TERMINAL '>'
#define QUOTE    '"'
#define ESCAPE   '\\'

void main(void)
{
    int c;
    while ((c = getchar()) != EOF)  {
        if (c == KEYTOKEN)  {
            long fileposition = ftell(stdin)-1;
            int counter = 0;
            putchar(QUOTE);
            while ((c = getchar()) != TERMINAL) {
                if (c == EOF || counter++ == MAXKEYLENGTH)
                    break;
                if (c == QUOTE)
                    putchar(ESCAPE);
                putchar(c);
            }
            putchar(QUOTE);
            putchar(',');
            printf("%ld\n", fileposition);
        }
    }
}
```

End Listing Four

Listing Five

```
/* ---------- keybuild.c ---------- */

/*
 * Build a test HyperTree from standard input.
 * This is pass three, which builds the HyperTree
 * from the sorted table.
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "hyprtree.h"

#define QUOTE    '"'
#define ESCAPE   '\\'

void main(void)
{
    int c;
    char key[MAXKEYLENGTH+1];
    char kv[20];
    KEYVALUE keyvalue;

    while ((c = getchar()) != EOF)  {
        if (c == QUOTE) {
            int counter = 0;
            char *cp = key;
            while ((c = getchar()) != QUOTE)    {
                if (c == EOF || counter++ == MAXKEYLENGTH)
                    break;
                if (c == ESCAPE)
                    c = getchar();
                *cp++ = c;
            }
            *cp = '\0';
            if (getchar() == ',')   {
                char *kp = kv;
```

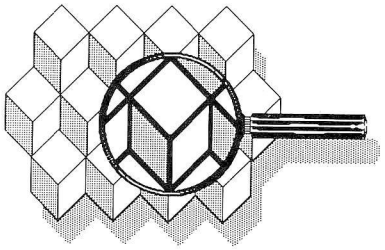
```
            while ((c = getchar()) != '\n' && c != EOF)
                *kp++ = c;
            *kp = '\0';
            keyvalue = atol(kv);
            addkey(key, keyvalue);
        }
    }
}
addkey(NULL, keyvalue);
}
```

End Listing Five

Listing Six

```
/* ------------ hyprsrch.c ------------- */

/*
 * Search the test HyperTree for a match on the
 * command-line parameter.
 * Display a screen of text starting at the
 * position indicated for the matching (or closest)
 * entry in the HyperTree.
 */

#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <conio.h>
#include "hyprtree.h"

#define SCRLINES 25

void main(int argc, char *argv[])
{
    if (argc < 3)
        fprintf(stderr, "Usage: hyprsrch textfile keyvalue");
    else    {
        int kb = 0;
        FILE *fp = fopen(argv[1], "r");
        if (fp == NULL) {
            fprintf(stderr, "No such file as %s", argv[1]);
            return;
        }
        findkey(argv[2]);
        while (toupper(kb) != 'Q')  {
            int lc = 4, c;
            KEYVALUE kv = current_keyvalue();
            printf("\n%s (%ld)", current_keystring(), kv);
            printf("\n-----------------------------------\n");
            if (kv) {
                fseek(fp, kv, SEEK_SET);
                while ((c = getc(fp)) != EOF && lc < SCRLINES){
                    if (c == '\n')
                        lc++;
                    putchar(c);
                }
            }
            printf("\nF = first, "
                   "L = last, "
                   "N = next, "
                   "P = previous, "
                   "Q = quit...");
            kb = getch();
            switch (toupper(kb))    {
                case 'F':
                    firstkey();
                    break;
                case 'L':
                    lastkey();
                    break;
                case 'N':
                    nextkey();
                    break;
                case 'P':
                    prevkey();
                    break;
                default:
                    break;
            }
        }
    }
}
```

End Listings



The Watcom C Optimizing Compiler and Tools has been upgraded to Version 8.0, **Watcom** announced. The company describes this as a complete line of professional C development products for the IBM PC and compatibles. Included in this version is support for OS/2 run-time libraries and Windows run-time conventions, and a 386 source-level debugger user interface provides multiple windows and mouse support. Watcom claims the compiler is better in such areas as performance, debugging capability, multiplatform support, language extensions, and performance tuning. And C8.0 is run-time compatible with the new Watcom Fortan 77 compiler.

Special versions of the compiler run in protected mode on OS/2 or 386 DOS systems and use memory greater than 640K to optimize large programs. A new execution profiler for performance tuning pinpoints high-use regions of code, so you can revise the parts of your program that get the most use. Expanded language support now includes IBM's SAA C specification, in addition to ANSI C and Microsoft C source compatibility. Both C8.0 and C8.0/386 are also available in professional editions, which provide such features as OS/2 and Windows support, the execution profiler, graphics library, and 386 protected-mode compiler. The 386 source-level debugger comes with the professional edition of C8.0/386. These editions cost $495 for C8.0 and $1,295 for C8.0/386. The standard editions cost $395 and $895, respectively. Reader service no. 20.
Watcom Products Inc.
415 Phillip St.
Waterloo, Ontario
Canada N2L 3X2
519-886-3700

A new hypermedia engine for manipulating multimedia information objects has been released by **OWL International**. They have redesigned their hypertext product, Guide; release 3.0 allows you to incorporate text, graphics, sound, and video images in hypermedia software for PCs.

DDJ met with William Nisen, the president of OWL, who explained that "Guide 3.0 has an object-oriented framework that was rewritten in C. The hypermedia engine manages objects that are directly addressable. Developers can customize the environment and keep the interface simple — one click of the mouse can do five things." He also said that in addition to providing the product, OWL provides the services to help people with the software.

A new full-function programming language, LOGiiX, gives you control over the Guide hypermedia engine, and allows you to create customized documents; hypermedia linking tools let you add multidimensional and versatile referencing capabilities; and you can import and export graphics through industry-standard file formats. Other improvements include hypertext links for searching interrelated or selected documents, document formatting for customizing the graphical user interface, text-in-window control of text placement and line wrapping, document maintenance tools, and Windows compatibility. Guide 3.0 costs $495. Reader service no. 21.
OWL International Inc.
2800 156th Ave. SE
Bellevue, WA 98007
206-747-3203

A pop-up file scan and hypertext tool for IBM PCs and compatibles, PC-Browse, is available from **Quicksoft**. This program lets you view files, find lost files, search files for information, and link them together, using hypertext. One hotkey press can, for example, import a text file into a spreadsheet application; once in view, it can either be printed or pasted into the host application. DOS wildcards allow searches in multiple files, directories, or drives, and the lookup search finds files with sorted records.

PC-Browse can be used for cross-referencing information; words can be flagged within whatever word processor you use, and then linked within files or between two or more files. You can customize on-line cross-references and make large files more manageable. You can also develop menu systems for end users, change hotkeys, reconfigure buffer size, and customize screen colors and delimiter characters. PC-Browse is shareware — full registration, for $49, includes the software, manual, one year of technical support, and a quarterly newsletter. OEM licensing is also available. Reader service no. 22.
Quicksoft Inc.
219 First Ave. N #224
Seattle, WA 98109
206-282-0452

Seminars on online documentation will be held this fall in Chicago, Boston, and San Jose, sponsored by **William Horton Associates**. These two-day seminars discuss issues such as the differences between paper and online documents; what should and should not go online; and various types of online documentation such as help, hypertext, videotext, hypermedia, and tutorials.

The seminars will also cover the styles online documents should be written in, options in display design, and the various software available for online documentation. The kinds of storage and delivery media, such as CD-ROM, file servers, and local- and wide-area networks will also be discussed, as well as how to make information accessible through menus, indexes, information retrieval, context sensitivity, and full-text search. The seminars will deal with problems such as producing online documents with limited staff and resources, converting existing paper documents to effective online documentation, and publishing both paper and online documents from a single source. Group activities and hands-on exercises, as well as handouts and William Horton's book *Designing and Writing Online Documentation*, are included in the $395 fee for the seminars. In-house seminars are also available. Reader service no. 25.
William Horton Associates
1523 Ward Ave. NE
Huntsville, AL 35801
205-536-8207

HyperEdit, a hypertext program editor from **SpeedyWrite Software**, can supposedly find any procedure, variable, or type, or anything else, anywhere in your program, no matter what language you use. HyperEdit features the ability to view an automatically generated list of variable, type, and procedure names (to search for parts of names or just scroll through the list); to look up calling sequences for C and Modula-2 library functions; to use wildcards in any search or look-up command; and to design complex searches including base conversion, arithmetic, and looking for characters in a certain range. HyperEdit also allows you to work with any number of files or languages in the same project.

A smart pair-matching command matches any parenthesis, bracket, brace, or language-specific pair such as begin and end, ignoring parentheses in string constants and comments, in order to check program balance. Included are such programming utilities as a calculator, ASCII table, screen-attribute table, and keycode display. Requires an

(continued on page 166)

(continued from page 161)
IBM PC or compatible with DOS 2.1 or later, and 384K of RAM. Hard disk recommended, lists for $69.95 for noneducational uses, $49.95 for educational (single user). Reader service no. 26.
SpeedyWrite Software
1600 Grand Ave., Box 1691
St. Paul, MN 55105
612-696-6732

Three hypertext-based reference tools are available from **Sageline**. Write*On is a comprehensive reference system that provides rules and tips on questions of grammar, usage, and punctuation. Effective examples illustrate the rules, and cross-references are provided when necessary.

GPOStyle is a hypertexted edition of the *Government Printing Office (GPO) Style Manual.* While we here at *DDJ* prefer *The Chicago Manual of Style,* GPOStyle is a handy reference to usage. It is not as comprehensive as Write*On, however.

The Shell automatically integrates existing and future Sageline reference systems, so that they are all accessible from one directory. It is packaged with the Almanac, a handy but limited reference. Categories include computers (character tables, numbers, PC extended key codes, and more), U.S. history (the Declaration of Independence, the Constitution and Amendments), language (the name implies a wider category — but one fun feature is the "Quotes for Contemplation"), math (useful formulas, logarithmic identifiers, geometry and trigonometry functions), science (physical constants, statistics on the Sun, Moon, and planets, a table of the elements), and geography (mail and UPS rates, state capitals, area codes, and time zones). Though a lot of work obviously went into the almanac, a more comprehensive treatment would be great.

All three are easy-to-use and well referenced. The only drawback is that you have to leave whatever application you're working in in order to access these tools. The Almanac costs $29, Write*On is $49, and GPOStyle is $79. Reader service no. 27.
Sageline
P.O. Box 2346
Kingston, NY 12401
800-345-5571

HyperTMON, a debugger for the HyperTalk programming language, is available from **ICOM Simulations**. The company claims this is the first tool of its kind for HyperTalk programming. It can find errors within HyperCard scripts and immediately modify them. The main purpose of HyperTMON is to enable users to study scripts step-by-step, shortening development time. Access to stacks is done simply by cutting and pasting the debug button from the HyperTMON stack. It sells for $99.95. Reader service no. 24.
ICOM Simulations Inc.
648 S. Wheeling Rd.
Wheeling, IL 60090
708-520-4440

If you're developing CD-ROM applications for Apple systems, you might be interested in the CD-ROM Developer's Lab from **Software Mart**. It is a multimedia production reference on CD-ROM — a searchable, full-text database that contains how-to information on all aspects of production. Includes such topics as design, programming, data preparation, transportability, media production, premastering, manufacturing, project management, encryption, and data assembly.

Included are functional applications that were created with Media-Mixer, another Software Mart product, which is a set of subroutine libraries for prototyping and creating full-text or multimedia CD-ROM databases and retrieval software. The tools for these libraries were written in Turbo Pascal and Microsoft C. The Developer's Lab also includes technical specifications, demonstrations of off-the-shelf tools for media production, and industry contacts for the fields of animation, sound production and editing, and high-resolution images. Requires a Mac with 1 Mbyte of RAM, a CD-ROM drive with an Apple-compatible SCSI interface, Apple CD-ROM driver v. 2.0 or later, and a printer. Retails for $795 (Media-Mixer usage licenses start at $1,750). Reader service no. 29.
Software Mart Inc.
4131 Spicewood Springs Rd., Ste. I-3
Austin, TX 78759-8606
512-346-7887

IBM LinkWay: Hypermedia for the PC, by Harrington, Fancher, and Black, has been published by **John Wiley & Sons**. It is a hands-on guide to understanding LinkWay, IBM's version of HyperCard, and is suitable for both beginners and experienced LinkWay users. It covers various components, object-oriented programming, and script commands. And it features a multimedia application to guide you through the process of combining media. ISBN 0-471-51298-2, $22.95. Reader service no. 30.
John Wiley & Sons
605 Third Ave.
New York, NY 10158
212-850-6000

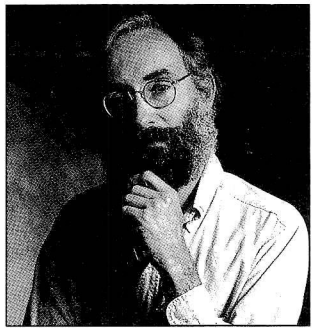
DDJ

Listing One *(Listing continued, text begins on page 126.)*

```
/* This is the main compression loop. Notice when the table is full we try
 * to increment the code size. Only when num_bits == MAX_BITS and the code
 * value table is full do we start to monitor the compression ratio.
 */
 while((character=getc(input)) != (unsigned)EOF) {
    if (!(++bytes_in % 1000)) {      /* Count input bytes and pacifier */
       putchar('.');
    }
    index=find_match(string_code,character);
    if (code_value[index] != -1)
       string_code=code_value[index];
    else {
       if (next_code <= max_code ) {
          code_value[index]=next_code++;
          prefix_code[index]=string_code;
          append_character[index]=character;
       }
       output_code(output,string_code);   /* Send out current code */
       string_code=character;
       if (next_code > max_code) {      /* Is table Full? */
          if ( num_bits < MAX_BITS) {     /* Any more bits? */
             putchar('+');
             max_code = MAXVAL(++num_bits);  /* Increment code size then */
          }
          else if (bytes_in > checkpoint) {         /* At checkpoint? */
             if (num_bits == MAX_BITS > max_code) {
              ratio_new = bytes_out*100/bytes_in; /* New compression ratio */
              if (ratio_new > ratio_old) {        /* Has ratio degraded? */
                output_code(output,CLEAR_TABLE); /* YES,flush string table */
                putchar('C');
                num_bits=INIT_BITS;
                next_code=FIRST_CODE;         /* Reset to FIRST_CODE */
                max_code = MAXVAL(num_bits); /* Re-Initialize this stuff */
                bytes_in = bytes_out = 0;
                ratio_old=100;                /* Reset compression ratio */
                for (i=0;i<TABLE_SIZE;i++)    /* Reset code value array */
                      code_value[i]=-1;
             }
             else                                /* NO, then save new */
             ratio_old = ratio_new;             /* compression ratio */
          }
          checkpoint = bytes_in + CHECK_TIME;    /* Set new checkpoint */
          }
       }
    }
 }
 output_code(output,string_code);   /* Output the last code */
 if (next_code == max_code) {       /* Handles special case for bit */
    ++num_bits;                     /* increment on EOF */
    putchar('+');
 }
 output_code(output,TERMINATOR);    /* Output the end of buffer code */
 output_code(output,0);             /* Flush the output buffer */
 output_code(output,0);
 output_code(output,0);
 putchar('\n');
}
/* UNCHANGED from original
 * This is the hashing routine.
 */
find_match(int hash_prefix, unsigned int hash_character)
{
   int index, offset;

   index = (hash_character << HASHING_SHIFT ) ^ hash_prefix;
   if (index == 0 )
      offset=1;
   else
      offset = TABLE_SIZE - index;
   while(1) {
      if (code_value[index] == -1 )
         return(index);
      if (prefix_code[index] == hash_prefix &&
                                 append_character[index] == hash_character)
         return(index);
      index -= offset;
      if (index < 0)
         index += TABLE_SIZE;
   }
}
/* MODIFIED This is the modified expansion routine. It must now check for the
 * CLEAR_TABLE code and know when to increment the code size.
 */
expand(FILE *input, FILE *output)
{
   unsigned int next_code=FIRST_CODE;
   unsigned int new_code;
   unsigned int old_code;
   int character,
   counter=0,
   clear_flag=1;          /* Need to clear the code value array */
   unsigned char *string;
   char *decode_string(unsigned char *buffer, unsigned int code);

   printf("Expanding\n");

   while((new_code=input_code(input)) != TERMINATOR) {
      if (clear_flag) {       /* Initialize or Re-Initialize */
         clear_flag=0;
         old_code=new_code;   /* The next three lines have been moved */
         character=old_code;  /* from the original */
         putc(old_code,output);
         continue;
      }
      if (new_code == CLEAR_TABLE) {     /* Clear string table */
         clear_flag=1;
         num_bits=INIT_BITS;
         next_code=FIRST_CODE;
         putchar('C');
         max_code = MAXVAL(num_bits);
         continue;
      }
      if (++counter == 1000) {           /* Pacifier */
         counter=0;
         putchar('.');
      }
      if (new_code >= next_code) {       /* Check for string+char+string */
         *decode_stack=character;
         string=decode_string(decode_stack+1,old_code);
      }
      else
         string=decode_string(decode_stack,new_code);

      character = *string;               /* Output decoded string in reverse */
      while (string >= decode_stack)
         putc(*string--,output);

      if (next_code <= max_code) {       /* Add to string table if not full */
         prefix_code[next_code]=old_code;
         append_character[next_code++]=character;
         if (next_code == max_code && num_bits < MAX_BITS) {
            putchar('+');
            max_code = MAXVAL(++num_bits);
         }
      }
      old_code=new_code;
   }
   putchar('\n');
}
/* UNCHANGED from original
 * Decode a string from the string table, storing it in a buffer.
 * The buffer can then be output in reverse order by the expansion
 * program.
 */
char *decode_string(unsigned char *buffer, unsigned int code)
{
   int i=0;

   while(code > 255 ) {
      *buffer++ = append_character[code];
      code=prefix_code[code];
      if (i++ >= 4000 ) {
         printf("Error during code expansion\n");
         exit(1);
      }
   }
   *buffer=code;
   return(buffer);
}

/* UNCHANGED from original
 * Input a variable length code.
 */
unsigned input_code(FILE *input)
{
   unsigned int return_value;
   static int input_bit_count=0;
   static unsigned long input_bit_buffer=0L;

   while (input_bit_count <= 24 ) {
     input_bit_buffer |= (unsigned long) getc(input) << (24 - input_bit_count);
     input_bit_count += 8;
   }
   return_value=input_bit_buffer >> (32-num_bits);
   input_bit_buffer <<= num_bits;
   input_bit_count -= num_bits;
   return(return_value);
}
/* MODIFIED Output a variable length code.
 */
output_code(FILE *output, unsigned int code)
{
   static int output_bit_count=0;
   static unsigned long output_bit_buffer=0L;

   output_bit_buffer |= (unsigned long) code << (32 - num_bits -
                                                    output_bit_count);
   output_bit_count += num_bits;
   while (output_bit_count >= 8) {
      putc(output_bit_buffer >> 24, output);
      output_bit_buffer <<= 8;
      output_bit_count -= 8;
      bytes_out++;                    /* ADDED for compression monitoring */
   }
}
```

End Listing

Hyperterminology from Hell



Hyperception *n* An altered state of consciousness induced by reading hypertext and characterized by the inability to focus on single, distinct ideas. Cognitive astigmatism

Hyperemptory *adj* Exceptionally abrupt, as a direct hypertext link to a random location in RAM

Hyperennial *n* Any topic of which computer journalists annually announce that this is The Year , such as Unix, networking, OS/2, AI, multimedia, desktop fill-in-the-blank, or hypertext

Hyperenthetical *adj* Characterized by being a digression within a digression (within a digression. . .) The variation in spelling is not arbitrary. The Indo-European root from which the par of parenthetical derives is spelled with an e, and means to grant reciprocally, with the idea of getting something back. The Indo-European tradition that one ought to be able to get something back, or just to get back, from a digression, perished with their culture

Hyperformance *n* Multidimensional ineptitude

Hyperfume *n* The smell of hype

Hypergonomics *n* An academic's idea of a catchy term

Hyperhaps *n* Goings-on in hyperspace

Hyperimeter *n* A multidimensional boundary separating the obvious from the irrelevant

Hyperiodical *n* Any nonlinear serial publication; a journal that appears regularly but not regular

Hyperipatetic *adj* Lost in hyperspace

Hyperipheral *adj* Lying beyond the hyperimeter, as opposed to ordinary lying

Hyperiscope *n* A hypertext navigational aid used when maps and browsers fail; in earlier days called a "core dump."

Hypermanent store *n* 1. The locus of data protected from accidental deletion by virtue of being lost; hypertext's contribution to the architecture of write-only memory. 2. A bouffant boutique

Hypermute *v.t* To rearrange hypertext links randomly. To engage in data annealing

Hyperparallel *adj* Skew

Hyperpendicular *adj* Skew

Hyperpetrate *v.t* To implement a hypertext system

Hyperpilosity *n* A measure of the hairiness of a hypertext system

Hyperplex *n* A movie theater of the 1990s. If, as has been suggested by no less eminent hypermedia experts than Ted Nelson and Paul Heckel, the future of the personal computer can be read on the silver screen, we should expect the workstation of 1999 to run more expensive software with less content and more flash, and to display it on six tiny monitors

Hypersian *n* Persian poet Omar Khayyàm, who wrote one of the oldest known nonlinear documents (later linearized by Edward Fitzgerald as the Rubaiyat) and left this advice to readers of hypertext:

Drink! for you know not whence you came, nor why;
Drink! for you know not why you go, nor where.

Hyperu *n* The Andes

Hyperversion number *n* A complexity measure for hypertext documents

Hypervert *interj* A greeting from one hypertext system designer to another

Hypuree *n* Hypertext with the links removed

Hypurgative *n* Garbage collection for hypertext

Hypurpose *n* A noble ambition worthy of significant financial backing but incapable of being expressed in terms that mere linearists can understand

Hypursuant *adj* In accordance with in a higher dimension, as in, "Hypursuant to your directive that the staff dress more formally while in the office, I am taking Friday afternoon off to go to the beach"

These hijinks were inspired by Stan Kelly-Bootle's *The Devil's DP Dictionary*, McGraw-Hill, 1981.

Michael Swaine

Michael Swaine
editor-at-large

#166 JULY 1990 $2.95 ($3.95 CANADA)

15 YEARS OF POWER PROGRAMMING

Dr. Dobb's JOURNAL

SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

Flying High With GRAPHICS PROGRAMMING

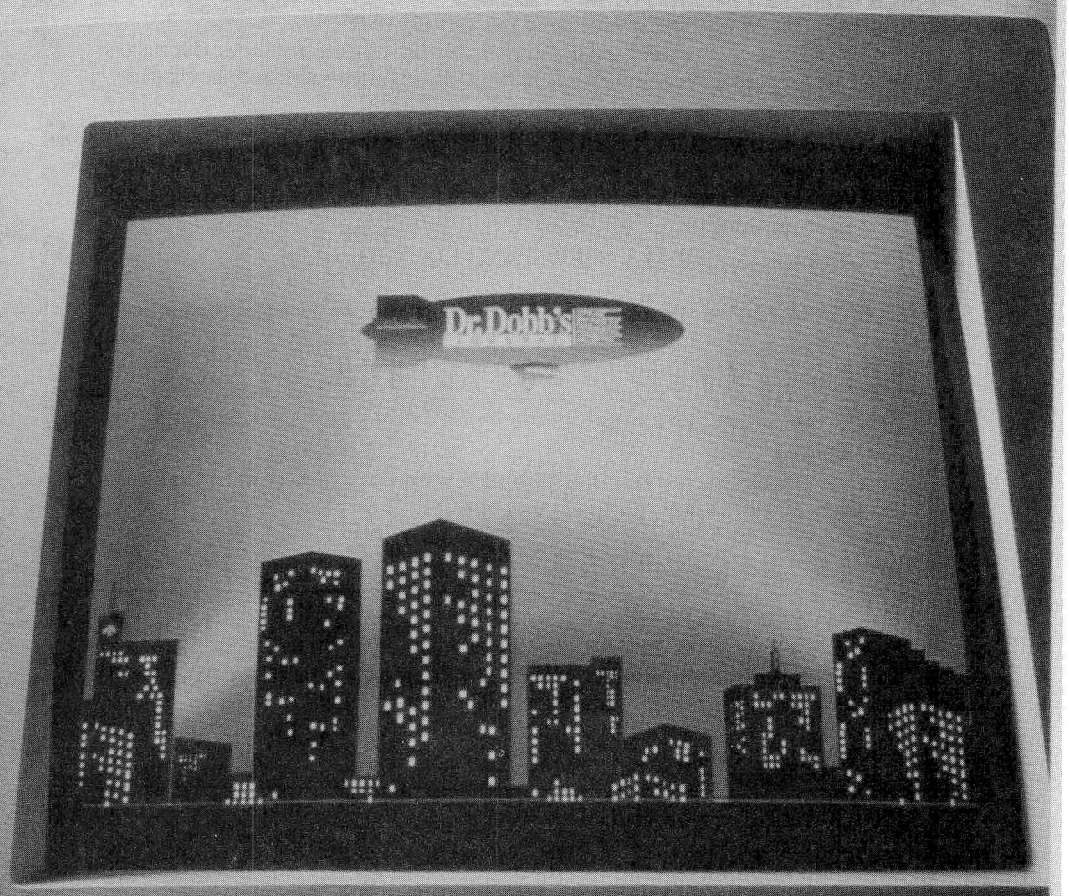
- STRAIGHTER LINES
- ROUNDER CIRCLES
- SMOOTHER CURVES

PROGRAMMING FOR SUPER VGA

SMALLTALK'S MVC

MORE POWER TO BASIC

ACCESSING 4 GIGABYTES IN REAL-MODE



SWAINE ON MACWORLD '90

0 38351 16562 8 07

CONTENTS

JULY 1990
VOLUME 15, ISSUE 7

FEATURES

SUPER VGA PROGRAMMING **16**
by Christopher A. Howard
Chris presents a VGA chip-set detection method, including functions for addressing video memory and displaying pixels at specified locations.

CIRCLES AND THE DIGITAL DIFFERENTIAL ANALYZER **30**
by Tim Paterson
The digital differential analyzer, which uses unscaled integers for drawing straight lines and circles, belongs in every graphics library.

IMPROVING LINE SEGMENT CLIPPING **36**
by Victor J. Duvanenko, W.E. Robbins, and Ronald S. Gyurcsik
Today's windowing systems demand high-performance line-clipping techniques. Our authors show how to squeeze more performance from the classic Cohen-Sutherland line-clipping algorithm.

DRAWING CHARACTER SHAPES WITH BÉZIER CURVES **46**
by Todd King
Todd examines and implements Bézier curves by using the literal rendering technique and the deCasteljau method.

INFORMATION MODELS, VIEWS, AND CONTROLLERS **54**
by Adele Goldberg
Smalltalk's Model-View-Controller architecture was built with user interface design in mind.

PROGRAMMER'S WORKBENCH

DOS + 386 = 4 GIGABYTES! **62**
by Al Williams
It really is possible to access the entire 80386 address space *in real mode* when you use the techniques Al presents here.

EXAMINING ROOM

THE POWER IN POWERBASIC **72**
by Bruce Tonkin
Can a leopard change its spots? Bruce finds out by examining Spectra's PowerBasic, previously known as Borland's Turbo Basic.

COLUMNS

PROGRAMMING PARADIGMS **123**
by Michael Swaine
Michael reports on MacWorld Expo, raises questions about Glasnost programming, and ruminates on recent issues in chaos theory, fractals, and neural networks.

C PROGRAMMING **131**
by Al Stevens
Al takes an early look at Turbo C++, a second look at ANSI C token pasting, and an in-depth look at hotkeys.

STRUCTURED PROGRAMMING **139**
by Jeff Duntemann
It's one darn thing after another as Jeff tackles object design and multiple inheritance.



A special thanks to Doc Livingston of RIX Software for helping with the image on the cover and to Robert Stephenson at CompuAdd for the 16-bit VGA card used to display it. Additional thanks to Eric Christenson at Western Digital for letting us use their VGA cards.

DEPARTMENTS

EDITORIAL **6**
by Jonathan Erickson

LETTERS **8**
by you

SWAINE'S FLAMES **160**
by Michael Swaine

PROGRAMMER'S SERVICES

ADVERTISER INDEX **152**
where to go for more information on products

OF INTEREST **153**
compiled *by Janna Custer*

PROGRAMMER'S MARKETPLACE **154**
classified ads

SOURCE CODE AVAILABILITY
All source code is available on a disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (inside Calif.) or 800-533-4372 (outside Calif.). Specify issue number and disk format. Code is also available through the DDJ Forum on CompuServe (type GO DDJ).

NEXT ISSUE

August brings our annual C issue, where we'll examine a variety of C programming topics, from memory allocation and debugging to linking C with other languages.

The Kent Porter Scholarship Fund

There's always been something special about *DDJ*'s coverage of graphics programming, this month's theme. As far back as July 1976 (our first year of publication), *DDJ* began publishing articles such as "A Home Brew TV Display with Graphics for the Altair 8800." From that point on, graphics programming became a regular topic in *DDJ*, leading up to Kent Porter's enormously popular "Graphics Programming" column in 1989. Sadly, Kent passed away before he had a chance to complete the job he set out to do.

In Kent's memory, we've established a scholarship program for full-time computer science majors enrolled in accredited colleges and universities. The purpose of the scholarship is to recognize academic achievement and potential, and to financially assist continuing students in the pursuit of their educational goals. At the request of Kent's family, consideration will be given (but not limited) to students who are raising children while attending school, a situation similar to that of Kent and Jeanne in their university days. Scholarships will be awarded in increments of $500 for the 1990 – 91 academic year.

M&T Publishing will provide $1,000 a year to fund the scholarship, and, for this year, Markt and Technik (M&T's parent company) will provide another $1,000. Additionally, we're accepting contributions to the scholarship fund from individuals and corporations. Your contributions will be appreciated and acknowledged if you take part in this program.

To apply for a scholarship, request in writing an application from:

The Kent Porter Scholarship
Dr. Dobb's Journal
501 Galveston Drive
Redwood City, CA 94063

Software being what it is, we made a couple of last minute changes to June's hypertext project. Because of the size of the self-extracting file (380K on the bare-bones version), we're providing the system as a PKZIP file; the resulting compressed file is about 275K. Once you extract either file, running the system is the same — just type DDJ.

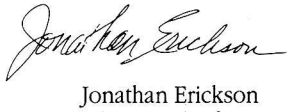
The hypertext document included with the June source code listings disk, available through M&T Books, is also compressed with PKZIP so that the issue will fit on a single 360K disk. PKUNZIP, which you'll need to extract the file, is included on the disk.

Remember that the enhanced version — the one that includes everything — is available online and from Scott Johnson at NTERGAID (and it's a *really* big file).

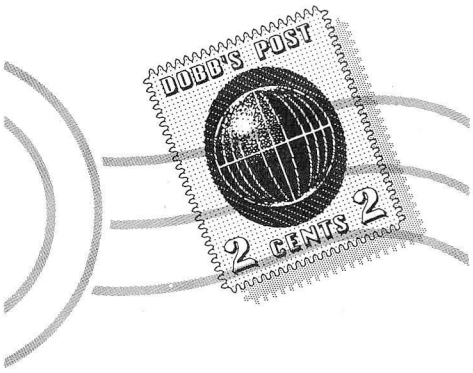
Once you've had a chance to use the hypertext edition, let us know what you think about it. What would you have liked to see different about it? How have you been using it? And, most importantly, what sort of applications will you be developing with the hypertext engine source code (and other programs) included in the issue?

Thanks for the great response to our May editorial survey card. We've received thousands of cards so far and more arrive every day. If you haven't returned your card, it isn't too late, so keep those cards and letters coming — and we really are reading every one of them.

Finally, the folks at Springtree Partners have compiled a *DDJ* subject and author index that spans the years 1982-1989. It's available on disk and in paperback. Call 804-286-3466, Ext. 3, or write Springtree Partners, Rt.2 Box 89, Scottsville, VA 24590-9512. They've done a great job of documenting the last seven years of *DDJ*.



Jonathan Erickson
editor-in-chief



Alive and Apparently Well in Louisville

Dear *DDJ*,

In his April 1990 "Structured Programming" column, Jeff Duntemann asked readers to drop him a postcard if they saw his book *Assembly Language From Square One*. Elvis was sighted browsing through a copy of *Assembly Language From Square One* in Hawley-Cooke Booksellers of Louisville, Kentucky.

David Rush
Louisville, Kentucky

A Plus for Patents

Dear *DDJ*,

I work in the research division of a major pharmaceutical company where I and my colleagues strive to invent new drugs. The research and development process for new pharmaceuticals typically requires seven years and averages $125 million. As such, patents are a necessary early part of the commercialization process. Without the exclusivity a patent grants, commercialization would be a far riskier venture.

My experience with patents is as an inventor. Although I am no legal expert, I do have a working knowledge of the patenting process. I also have a perspective on the patent issue brought on both by my vocation and by an involvement with software development. I would like to clarify some issues raised in your March 1990 editorial.

The enormous hue and cry generated by the issue of software patents has left me puzzled as to the reasons behind the concerns. I believe this reason is a fear that the access to ideas will be prevented in an unfair manner, the result of which is stifled innovation. Actually, the real cause is a fundamental misunderstanding of the purpose of patents and how they should fit into software development. A component of the problem is the apparent suddenness at which patents are issued. The solution to the problem will be to expose the details of software patents and educate the professional programming community accordingly. They will have to understand that the long delay between filing and issuance is part of the patent process and builds in risk for those who use patentable ideas without having patent protection. Programmers may not like software patents, but they should also know that software patents will not go away.

Algorithms, as such, are not patentable. Algorithms that have an application can be considered inventions and are patentable. In short, any idea which can be shown to be novel and useful can be patented. An issued patent gives the inventor a period of exclusive control over the invention, which, as memory serves, is 17 years, in exchange for full disclosure of the details of the invention. After the exclusivity period is over, the inventor has no control and anyone is free to use the invention. A valid application is an improvement over existing inventions, referred to as "prior art." As such, the first electronic spreadsheet may have been patentable as an invention (although it is "obvious") because of the improved performance. General patents that cover specific patents are considered to "roof" the specific patents and restrict the use of the application by the inventor. An example of roofing would be a general patent on the basic concepts of a spreadsheet — it would restrict the application of a patent for automatic recalculation of spreadsheet cells. Finally, patents make general and specific claims as to what constitutes the invention, the so-called scope of the invention. For a software patent, any new idea not previously claimed for a patent is in itself patentable. If the LZW data compression algorithm was claimed to be only useful for telecommunications, the same algorithm could be patented for, say, archival database data compression and Unisys could do nothing about it.

As for litigation, the patent law is stacked entirely on the side of the inventor. There is nothing necessary about needing to outwait those with less resource: if the infringement is clear (an important point), the infringer loses. The problem here is not so much willful infringement, which is really intellectual property theft, but confusion caused by the delay in patent issuance. Programmers are quick to adopt good ideas, some of which constitute patentable inventions, with the result that several products may emerge into the marketplace before the patent is issued to one company — and the others lose. This gives the impression of restricting the market and can be dissatisfying to consumers. One company, whose name has escaped me, has even gone as far as sending letters to the owners of a competing product to inform them that it was about to be rewarded with the patent covering the product, and all those who had the other products would be liable for damages. Now that is disgusting and should be discouraged. The solution to this is for companies and programmers to be more patent aware. This could be accomplished if those filing for patents publicly state their intent, thus forewarning others. Could it happen? Probably not, because lawyers control the dissemination of product development information, the nature of which is typically confidential.

Does patenting stifle innovation? No, not at all. Only those working on comparable projects are affected and, naturally, the use of the algorithm for the claimed applications is protected in all its forms. This is far better protection than one can get by copyrighting code, which only protects the expression of the algorithm, not the actual algorithm. Therefore, if someone invents a novel algorithm, one with commercial potential, patent protection is the proper course. The use of patented inventions is usually regulated by lawyers in the form of licensing and royalty arrangements. If a patented algorithm is essential to the success of another software project, the appropriate legal arrangements can usually be made to use the invention and still make money.

Does the exclusivity time of 17 years stifle innovation? No. Patents are subject to technology changes, just as everything is. If the Basic interpreter that was *DDJ's* charter project 15 years ago had critical elements patented, the exclusivity would still have two years to go. Just looking in the pages of *DDJ* over the last seven years, one wouldn't know Basic existed. The facts are that times change and patents can become obsolete. However, if some lucky inventor hit the right idea, one that consumers wished to consume, he would have the luxury of no competition, could enjoy the fruits of his labors, and perhaps even improve the invention. Does that mean he can charge whatever he wants and the consumers must pay? No. Patent rights do not translate to automatic sales. The natural forces of the marketplace are always at work. If an inventor asks too high a price, he quickly finds that most consumers can live without his invention. The result is that patents do not free the inventor from normal marketing considerations,

(continued on page 12)

(continued from page 8)
but do remove the competition for a time.

An issue that will affect software patents, both in review time and quality, is an apparent lack of patent examiners with expertise in the area of software patents. With few examiners, the review time increases from two years to many years. If their expertise isn't fully up to snuff, poor quality patents can slip through. Poor quality can mean many things, ranging from the actual invention to the scope of the invention, which would cover more general applications of the invention.

Personally, I would like to see those who are complaining loudly go off and invent something new. The world needs perhaps one less spreadsheet program or database and perhaps more radical, innovative, and new programs. But we will never know what those new programs are until someone invents the fundamental algorithms to power them. I would also like those who do manage to patent new software inventions to do the next step, which is to bring that invention to consumers in the form of running programs. That is the intent of the patent law (which is as old as this country) — to encourage inventors to invent and bring those inventions to market. It was good then, it is just as good now.

My bottom line is that programmers should consider an issued software patent an opportunity to work another area, and be inventive in their own right. Patents should provide protection for the inventor, and patented inventions should be brought to market. And some form of information exchange needs to be developed so that those applying and receiving patents can let their intentions be known.

Barr Bauer
Bloomfield, New Jersey

Here We Go Again

Dear *DDJ*,

In your assembly language issue (March, 1990) you do your readership a disservice. You allow Michael Abrash to propagate the myth that code produced by compilers cannot match code produced by good assembler language programmers. The truth is the compilers used in his comparison are not worthy of being called "optimizing."

Consider his example of CopyUppercase. A C programmer might write the body as that in Example 1. I would expect any compiler to construct the intermediate form in Figure 1. I would then expect an optimizing compiler to identify common subexpressions and perform dataflow analysis resulting in that in Figure 2. During memory allocation I would expect an optimizing compiler to recognize that the idioms in Figure 3 can be implemented using string instructions if the DS,SI and EI,DI registers are associated with the appropriate subexpressions. Without such recognition, string instructions will never be generated. Recognition of these idioms leads to the allocation for subexpressions to machine locations as in Example 2, which leads to the code shown in Example 3. Since these ideas

(continued on page 14)

```
do {
  x = *a++;
  x += ((('a'<=x)&&(x<='z'))? ('A'-'a'): 0;
  *b++ = x;
} while (x);
```

Example 1

subexpression name	register name	register contents
@a,offset	DS,SI	address for source byte
@b,offset	EI,DI	address for destination byte
x, x+0, x+32	AL	source/destination byte value

Example 2

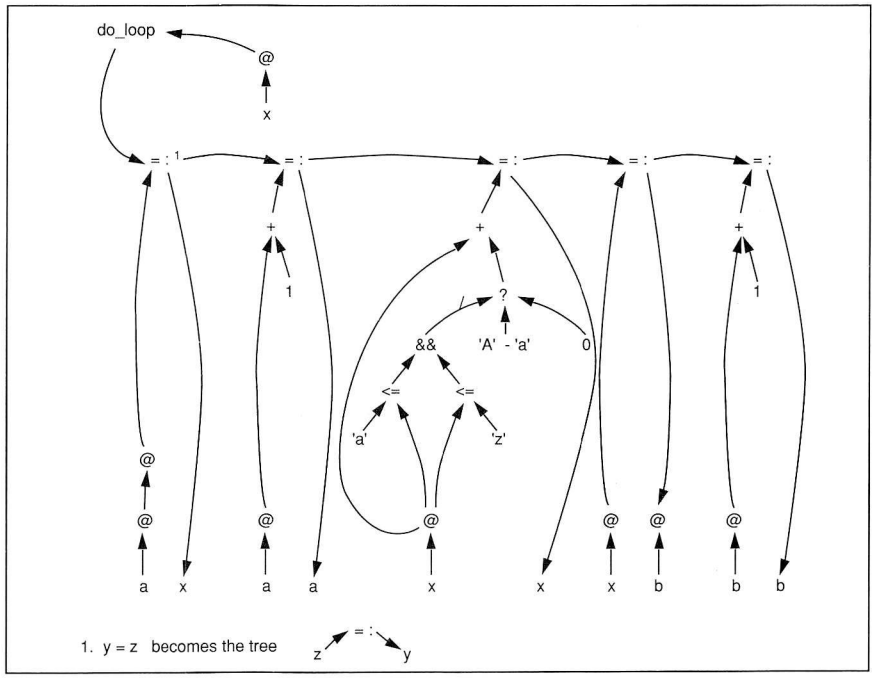


Figure 1

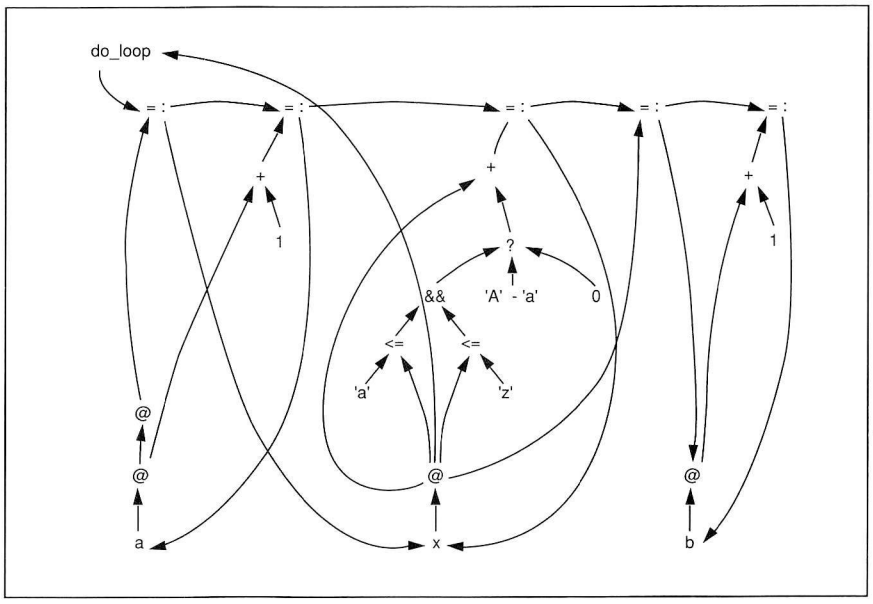


Figure 2

(continued from page 12.)
are at least 10 – 15 years old, any optimizing compiler should incorporate them. Michael Abrash made a mistake in regarding the Microsoft and/or Borland compilers to be optimizing compilers. By the way, I am The OPG Co., a firm that performs research into compiler writing tools.

George H. Roberts
Broken Arrow, Oklahoma

Michael responds: And so once again we come to the difference between "what is" and "what should be." Mr. Roberts' letter reminds me of the debate between the RISC and CISC people; the RISC people keep saying that RISC has the potential to be two, four, even ten times faster than CISC, and the CISC people keep sighed and pointing out that today's CISC software, running on today's CISC computers, is just about as fast per dollar cost — and there's a heck of a lot more of it available. In the high-level versus assembly language arena, it's much the same. I remember a lively debate three or four years ago about the relatively low quality of code generated from C source, with the C proponents insisting that the critics were mistaking poor compiler implementations for a poorly optimizable language. "Just wait until real optimizing C compilers arrive!" they protested.

Well, here we are, years later, and 95 percent of the world uses two compilers that Mr. Roberts claims aren't optimizing compilers at all. I note that Mr. Roberts doesn't actually name a compiler that generates the code he lists; even if there is such a compiler, I suggest that since almost no one is using that compiler, whatever it is, it's a moot point. Mr. Roberts may be entirely correct in that a good optimizing compiler will generate the code he lists; I suggest that today, given currently used PC tools, that's pretty much irrelevant. There's another point to be addressed here. After claiming that I "propagate the myth that code produced by compilers cannot match code produced by good assembler [sic] language programmers," Mr. Roberts follows with the nonsequitur that the compilers I used aren't worthy of being called optimizing compilers, as if the latter is evidence for the former. Even if Turbo C and Microsoft C aren't optimizing compilers, that doesn't mean that "worthy" compilers can generate code as good as assembly language programmers can. Consider Mr. Roberts's own example: His hypothetical "optimized" code is indeed much better than the code Microsoft C produced — but it's a good 50 percent slower than the hand-optimized code in my article! Anyone who believes that a compiler can match top-notch assembly language for small, well-defined tasks is kidding themselves. (There just isn't enough information bandwidth from the programmer to the compiler in a high-level language for this not to be true.) Assembly language isn't appropriate for most tasks, but when you need maximum performance, it is the only choice.

```
        ...         optimized prologue
        les         si, [bp+a_pointer]
        les         di, [bp+b_pointer]
Convert_and_Copy_Loop:
        lodsb
        cmp         al, 'a'
        jb          Save_Upper
        cmp         al, 'z'
        ja          Save_Upper
        add         al, 'A'-'a'
Save_Upper:
        stosb
        and         al,al
        jnz         Convert_and_Copy_Loop
        ...         optimized epilogue
```

Example 3

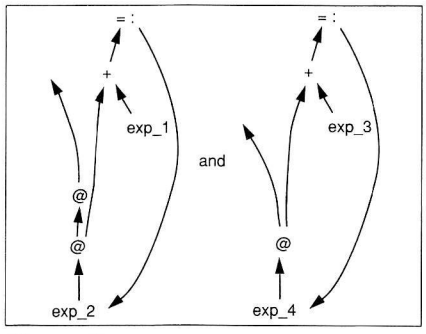


Figure 3

Errata

Please note the following changes to the source code listing on page 86 in last month's "Building a Hypertext System" by Rick Gessner (*DDJ*, June 1990).

```
44:  (( '┌','┐','└','┘','-','-','|','|'),
45:   ( '╔','╗','╚','╝','=','=','‖','‖'));

112: If Line [j]<>Null then
113:       Inc(i) else Inc(j,2);
114:     Inc. (j);
115: end;
116:  Determine_Actual_Line_Pos:=J;
117: end; (Determine actual line pos)

140: If Ord(Line[LinePos])>127 then
141: Begin
```

On page 167 of the same issue, change the line of source code accompanying "LZW Revisited" by Shawn Regan from:

```
if ( num_bits == MAX_BITS > max_code ) {
```

to

```
if ( num_bits == MAX_BITS ) {
```

DDJ apologizes for the confusion.

DDJ

Super VGA Programming

Coping with a myriad of options

Christopher A. Howard

Once upon a time users were presented a short and simple list of display adapter choices — color or monochrome. Although, from a programmer's perspective those days are still enticing — a myriad of standards has developed since then. The resolutions of the Hercules, CGA, EGA, and VGA adapters have all provided a common ground for users and developers alike. The next step (as defined by IBM) is the 8514/A, but it has been slow in its adoption. There is also Texas Instruments TIGA (pronounced TIE-GA), an interface for its 34010 and 34020 chips, but it, too, is in the first stages of support. Meanwhile, a number of video board manufacturers have released adapters with capabilities beyond the IBM VGA standard resolution of 320 × 200 × 256. These extended resolutions are collectively known as Super VGA.

In the simpler days of CGA and Hercules, the graphics screen was directly memory mapped. All a programmer had to do was point at it and write it. Aside from the minor headache of interleave addressing, graphics programming was relatively easy. To save the screen, just save that block of memory with a function such as BSAVE, and load it with BLOAD. Later came the EGA with its planar memory and hosts of registers, and things suddenly got more complicated and easier at the same time. Although functions such as BSAVE would no longer work, features like masking and logical operations became easier. The VGA followed this theme with higher resolutions and more colors, and finally we have Super VGA.

This article provides the ability to program the Super VGA modes for some of the major chipsets. A method of detecting each of the chipsets is presented, along with functions for addressing video memory and displaying a pixel at any location. Because any graphics function reduces down to plotting a pixel somewhere on the display, it is possible to extrapolate these functions to any other graphics operation, from the simple to the complex.

Chris is president of Genus Microprogramming, which provides a number of tools that support Super VGA (including the PCX Programmer's Toolkit, PCX Effects, and PCX Text). He can be reached at Genus Microprogramming, 11315 Meadow Lake, Houston, TX 77077, or on CompuServe: 75100,17.

The Chipsets

Currently, there are many chipsets on the market that provide Super VGA capabilities. In order to provide an adequate sample, I'll cover three of the major chipsets — Tseng Labs, Paradise, and Video Seven. Although board vendors may use the same chipset, it is still up to the vendor which features are incorporated into a particular adapter (see Table 1). Super VGA offers the advantages of higher resolution and color capabilities, with the disadvantage of no existing standard. In fact, each chipset must be programmed separately even though you may wish to support only one mode, such as 640 × 480 × 256. Even something as basic as mode numbers can vary wildly between board manufacturers.

Identification

Some manufacturers have made it easy to identify their chips. One of the easiest is Video Seven. Basically, they have provided an extended BIOS function call that returns valid information when the chipset is present, and garbage otherwise. This provides a consistent and reliable interface.

The other vendors, however, provide no method of identifying their chips. In these cases, a crude but effective BFAI (brute force and ignorance) approach can be used. All vendors include a copyright notice at the beginning of the video BIOS. You can search this area for strings — such as "Tseng" or "PARADISE" — in order to identify the chip. This method has worked rather well, but I would like to take this opportunity to suggest that these vendors provide a solution similar to Video Seven's.

This method is not a new one. Many software vendors used a similar technique in order to identify an IBM adapter, and they keyed in the letters "IBM" in the copyright string.

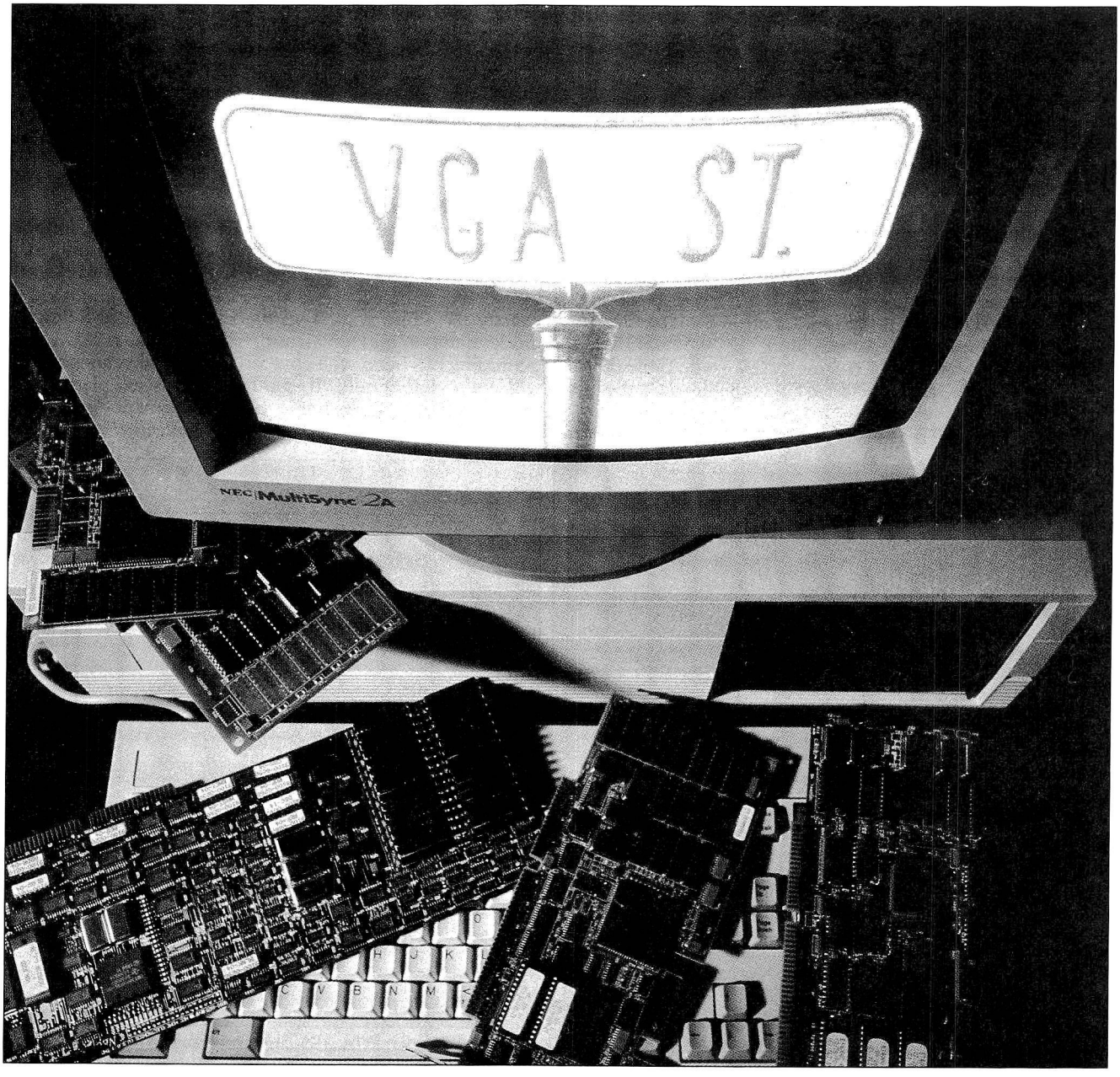
This, of course, caused problems for clone vendors. How could they remain compatible with those software packages, and still use the letters IBM without copyright conflicts? Orchid Technology's solution is shown in Figure 1, but for fun you can use DEBUG to display your video board's BIOS by running DEBUG and using the command *D C000:0,100* to dump the first 256 bytes of your VGA's ROM BIOS.

Initially, a function should be called that identifies the standard adapters (such as HGC, CGA, EGA, or VGA), but that would cover another article. An excellent reference is Richard Wilton's book, *Programmer's Guide to PC and PS/2 Video Systems* (Microsoft Press). Once a VGA adapter has been identified, the *svQueryChipset* function can be called to identify whether the VGA has a chipset that supports Super VGA resolutions. It tests for boards that have a BIOS function call first, and continues with ROM BIOS searches if that fails. In the event no match is found, it returns an error code. At that point your program must query the user, or assume that no supported Super VGA adapter exists.

Initializing the Mode

Initializing the Super VGA graphics mode is no different from setting any standard graphics mode. Each adapter has extended the ROM BIOS interrupt *10H* function so that the Super VGA modes are included. However, each board vendor has assigned its own mode numbers to those extended modes — which is one reason why they are harder to support. Because there is no standard mode number, a table of modes must be maintained for all supported chipsets.

The example code uses a two-step process for managing display modes, based on the method Genus Microprogramming uses for our graphics products. First, the display type is set with the function *svSetDisplay*, which selects the chipset and the mode. A defined constant is used so that other display types and modes can be easily added, thereby hiding the internal workings of the library. Note that this function does not actually set the mode — it only performs initialization internal to the Super VGA library and does not affect the calling program's environment. The mode change is performed in the *svSetMode* function, which takes as its



arguments either the *svTEXT* or *svGRAPHICS* constants. Listing One, page 82, and Listing Two, page 84, contain the defines macros for Super VGA graphics manipulation. Both are implemented in Microsoft ASM, Version 5.x. The file in Listing Three, page 84, contains procedures for identifying various Super VGA adapters and uses the procedures *svQueryChipset*. Listing Four, page 85, contains the internal routine procedures for calculating a pixel's address for any given display mode using the procedures *svPixelAddr2D*, *svPixelAddr30*, and *svPixelAddr5F*. Listing Five, page 86, contains procedures for putting (displaying) a pixel for any given display mode or virtual buffer using the procedure *svPutPixel*. Listing Six, page 90, lists the function declarations for the Super VGA library for C, while Listing Seven, page 90, is a Microsoft C 5.1 test program for testing the Super VGA *QueryChipset* and *PutPixel* functions. (Compile with *CL /c /AS svTest.C* and link *svTest , , , svLib;*). Listings Eight (page 91) and Nine (page 92) are the make files for making the Super VGA Library and the Super VGA test program. For more detail on Super VGA used, Table 2 lists the ports, indexes, and functions. Figure 3 illustrates the bits required for Bank selection.

Initializing the Super VGA graphics mode is no different from setting any standard graphics mode. Each adapter has extended the ROM BIOS interrupt 10H function so that the Super VGA modes are included

There are several reasons for splitting up the initialization and mode change functions. First, it allows the library to coexist with other graphics libraries, each of which requires an initialization routine of its own — and its routine needs to set the mode. In that case, a call to *svSetDisplay* and then its *init* would set the mode and make both libraries usable. Second, most initialization is required only once, not every time the mode is set. By calling *svSetDisplay* once, later calls to *svSetMode* can be made to switch back and forth between graphics and text modes. Lastly, the program is easier to maintain because the initialization function is performed once at the top of the program, not every time the mode is changed.

What you may notice is that the mode numbers in the table for the Video Seven chipset do not match the published mode constants. Technically, the correct modes are the *66H*, *67H*, and *69H* numbers, and Video Seven recommends that the mode be set through a modified BIOS *SetMode* function. We found that every time the mode was set to *66H*, a BIOS *GetMode* function call would always return *1AH*. This caused problems because the *svSetMode* function always reads back the mode to ensure it was successfully set — and it would always appear that it did not. To get around this problem and provide a standard way of setting and getting the mode, all of our libraries just go ahead and set the mode to *1AH* (or *1BH* or *1DH*), and everything works fine. If you wish to set the mode for the Video Seven in the recommended way, use the *@V7Mode* macro provided.

Extended 16-Color

There is one Super VGA mode that remains fairly constant across all boards, the 800 × 600 × 16 color mode. It is basically an extension of the 640 × 350 × 16 color EGA and 640 × 480 × 16 color VGA modes. It is programmed in exactly the same way, with the major changes being the mode number and pixel addressing. Because EGA programming has been well covered for several years now, we will not go into it here. To summarize, it is still planar, with each of the four planes fitting into one segment (64K bytes),

(continued on page 22.)

Chipset	Adapter	800 × 600 × 16	640 × 350 × 256	640 × 400 × 256	640 × 480 × 256	800 × 600 × 256
Tseng Labs	Orchid ProDesigner VGA	29H				
	Orchid ProDesigner Plus	29H	2DH		2EH	30H
	Genoa 5300	29H				
	Genoa 5400	29H	2DH		2EH	30H
	STB Extra/EM	29H	2DH		2EH	30H
	Tseng	29H	2DH		2EH	30H
Paradise	Paradise Plus-16	58H				
	Paradise Professional	58H		5EH	5F	
Video Seven	Video Seven Fastwrite	62H		66H		
	Video Seven VRAM	62H		66H	67H	69H

Table 1: *Video adapters and supported modes*

```
C000:0000  55AA30EB  5B546869  73206973  206E6F74   This is not
C000:0010  20612070  726F6475  6374206F  66204942   a product of IB
C000:0020  4D202028  49424D20  69732061  20747261   M (IBM is a tra
C000:0030  64656D61  726B206F  6620496E  7465726E   demark of Intern
C000:0040  6174696F  6E616C20  42757369  6E657373   ational Business
C000:0050  204D6163  68696E65  7320436F  72702E29   Machines Corp.
C000:0060  EB6F202A  20436F70  79726967  68742863   Copyright ©
C000:0070  29313938  38205473  656E6720  4C61626F   1988 Tseng Labo
C000:0080  7261746F  72696573  2C20496E  632E2030   ratories, Inc. 0
C000:0090  382F3039  2F383820  56382E30  3058014F   8/09/88 V8.00X O
C000:00A0  72636869  64205465  63686E6F  6C6F6779   rchid Technology
C000:00B0  20496E63  2EAB4400  C0000000  00000000   Inc.
```

Figure 1: *Orchid ProDesigner VGA ROM-BIOS dump*

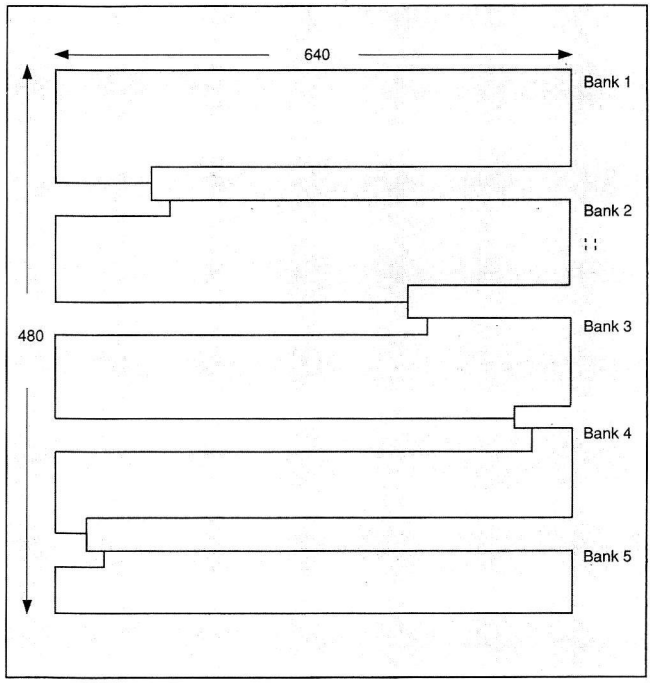


Figure 2: *Super VGA 256-color memory layout for 640× 480× 256 mode*

(continued from page 18)
for a total memory requirement of just under 256K bytes. To address a single pixel, use the formula *b = ((y*800)+ x)/ 8*, then mask out the pixel in that byte. The bit you want to affect is *x* mod *8*. All logical operations and bit masking are performed as any normal EGA or VGA 16-color mode.

Extended 256-Color

The Super VGA 256-color modes are an entirely new format. The memory layout is different from any previous video mode. It is not interleaved such as the CGA or Hercules adapters, nor is it planar such as the EGA and VGA 16-color modes. It is closest to the 320 × 200 × 256 VGA *13H* mode, complicated by the fact that the memory crosses 64K-byte segment boundaries.

Like the EGA and VGA, the Super VGA video memory is accessed through a 64K-byte area located at A0000 hex. To get around the memory requirements of the Super VGA modes (a mode like 640 × 480 × 256 requires over 300K of video RAM), all chipsets use a sliding window scheme. For the Tseng and Video Seven chipsets, this window can be thought of as a 64K bank of memory that can be located at any 64K boundary. Do not think of it as a purely rectangular window — the 64K window can start and stop anywhere within video memory (see Figure 2). This also means that the window starts and stops in the middle of a scanline, limiting some optimizations (such as checking for a bank change at the end of a scanline instead of any pixel). For the Tseng chipset, bank selection is all handled within a single byte. However, the Video Seven chipset gets the award for the most complicated bank switch. It involves setting three different bits in three different registers. The bank selection code for the example library is performed through macros,

(continued on page 26)

Port		Index		Function		Description
EGAgraph	(3CEH)					EGA/VGA/SVGA graphics controller
		parPROA	(09H)	(bank)		Paradise PROA reg used for bank sel
		parPR5	(0FH)	parLOCK	(0H)	Paradise PR5 reg used for locking and unlocking
				parUNLOCK	(5H)	
EGAseq	(3C4H)					EGA/VGA/SVGA Sequencer register
		v7pagesel	(F9H)			
		v7banksel	(F6H)			
		v7SR6	(06H)	v7enable	(EAH)	Video7 SR6 reg used for locking and unlocking
				v7disable	(AEH)	
VGAsegsel	(3CDH)			(bank)		VGA Segment select (Tseng)
VGAmisci	(3CCH)					VGA Miscellaneous In
VGAmisco	(3C2H)					VGA Miscellaneous Out

Table 2: *Register explanations*

Dr. Dobb's JOURNAL
SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

PUBLISHER *Peter Hutchinson*

EDITORIAL
EDITOR-IN-CHIEF *Jonathan Erickson*
MANAGING EDITOR *Monica E. Berg*
TECHNICAL EDITORS *Michael Floyd, Ray Valdes*
ASSOCIATE MANAGING EDITOR *Janna Custer*
CONTRIBUTING EDITORS *Al Stevens, Jeff Duntemann, Martin Tracy, David Betz, Tom Genereaux, Andrew Schulman*
COPY EDITORS *Rhoda Simmons, Pamela Dillehay, Nan Fornal*
EDITOR-AT-LARGE *Michael Swaine*

ART/PRODUCTION
ART/PRODUCTION DIRECTOR *Larry L. Clay*
ART DIRECTOR *Michael Hollister*
PRODUCTION SUPERVISOR *Amy Shulman Lesovoy*
TYPOGRAPHERS *Teresa Raines, Margaret Anderson, Charlene Carpentier*
COVER PHOTOGRAPHER *Michael Carr*

CIRCULATION
DIRECTOR OF CIRCULATION *Maureen Kaminski*
CIRCULATION MANAGER *Randy Robertson*
CIRCULATION PLANNING MANAGER *Manny Sawit*
DIRECT MARKETING MANAGER *Andrea Weingart*
NEWSSTAND MANAGER *Sarah Forsman*
DIRECT MARKETING COORDINATOR *Susan Bauman*
PROMOTION COORDINATOR *Philip Tsang*
FULFILLMENT COORDINATOR *Anne Jean*

ADMINISTRATION
VICE PRESIDENT OF FINANCE *Kate Deschamps*
CONTROLLER *Mary Collopy*
CREDIT MANAGER *Betty Arsene*
ACCOUNTING SUPERVISOR *Renate Kernke*
ACCOUNTS RECEIVABLE *Wendy Ho*
ACCOUNTS PAYABLE *LuAnn Rocklewitz*

MARKETING/ADVERTISING
DIRECTOR OF SALES AND MARKETING *Karla Spormann*
ADVERTISING COORDINATOR *Laura Stack Pullen*
MARKETING ASSISTANT *Sara Noah Ruddy*
ACCOUNT MANAGERS *see page 152*

M&T PUBLISHING INC.
CHAIRMAN OF THE BOARD *Otmar Weber*
DIRECTOR *C. F. von Quadt*
PRESIDENT *Laird Foshay*
VICE PRESIDENT OF PUBLISHING *William P. Howard*
VICE PRESIDENT/GROUP PUBLISHER *Randall L. Stickrod*

DR. DOBB'S JOURNAL (USPS 307690) is published monthly, except semimonthly in December, by M&T Publishing, Inc., 501 Galveston Dr., Redwood City, CA 94063; 415-366-3600. Second-class postage paid at Redwood City and at additional entry points.

ARTICLE SUBMISSIONS: Send manuscripts and disk (with article, listings, and letter to the editor) to the associate managing editor 415-366-3600.

***DDJ* ON COMPUSERVE:** Type GO DDJ.

SUBSCRIPTION: $29.97 for 1 year; $56.97 for 2 years. Foreign orders must be prepaid, including the additional postage (air or surface) in U.S. funds drawn on a U.S. bank. Add $13 per year for surface mail; add $36 for airmail to Canada and Mexico; or $26 for airlift to all other countries.

POSTMASTER: Send address changes to *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. **ISSN 1044-789X**

CUSTOMER SERVICE: For subscription questions, call toll-free 800-456-1215. For subscription orders or change of address call toll-free 800-456-1215 (U.S. and Canada) or write *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. For book/software orders call 800-533-4372 (in California 800-356-2002).

FOREIGN NEWSSTAND DISTRIBUTOR: Worldwide Media Service Inc., 115 E. 23rd St., New York, New York 10010; 212-420-0588 FAX 212-420-1265.

Entire contents copyright ©1990 by M&T Publishing, Inc., unless otherwise noted on specific articles. All rights reserved.

The Audit Bureau

(continued from page 22)
called *@TsengSeg* and *@V7Seg*.

The Paradise chipset is a little different in that it does not restrict the window to 64K boundaries. It is closer to an actual sliding window because it can be placed at any 4K increment. Unlike the Tseng and Video Seven, the window can be kept purely rectangular by placing it every 32 lines.

Each chipset must be programmed separately even though you may wish to support only one mode, such as 640 × 480 × 256. Even something as basic as mode numbers can vary wildly between board manufacturers

This is because 32 lines * 640 bytes = 20480 = 5 * 4K increment. For some functions this is an advantage, but in the case of *svPutPixel* it does not help. The macro *@ParSeg* emulates the other chipsets by forcing the window to a 64K increment (every sixteenth 4K position).

Note that both the Video Seven and the Paradise chipsets require that the extended Super VGA registers be unlocked/enabled before they can be written to, and they recommend that these registers be locked/disabled when the function

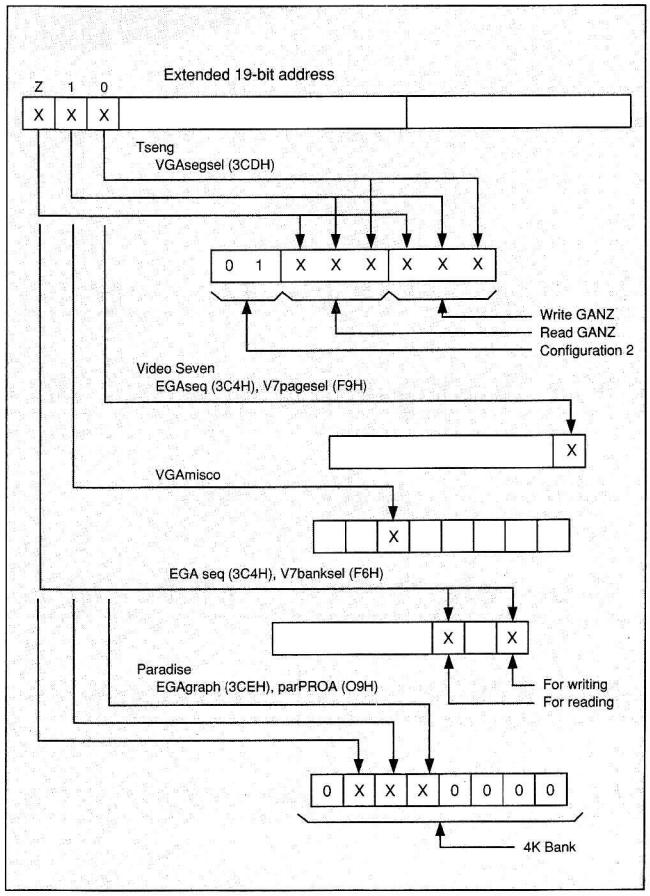


Figure 3: *Bank selection register descriptions*

(continued from page 26)
has completed using them. This is done with an *out* to a port, and they are supposedly locked in order to prevent accidental writes to those port locations. The only extended registers used in these examples involve bank selection.

Addressing Memory

Address calculations are simpler in the 256-color modes because each pixel is represented by 1 byte, and the memory is linear (all in a row, with no planes, and no interleaving). Compared to the CGA and Hercules interleaving and the planes of the EGA and VGA, Super VGA is the easiest.

The address calculations are performed by the functions in the svPA.ASM module (see Listing Four). Only two functions are necessary: One for 640-wide modes, and one for 800-wide Tseng and Video Seven modes. The routines merely involve multiplying the *y* coordinate by the screen width, and then adding in the *x* coordinate. Because each pixel is a byte, no conversion is necessary. The 64K bank number is automatically determined by the multiplication, because the AX register holds a maximum of 64K before it overflows into DX. Thus, the bank is returned to DX and the offset into that bank is returned in AX — easy.

Displaying a Pixel

Now that the pixel can be addressed, it is only a matter of writing it. The *svPutPixel* function contained in the svPP.ASM module (see Listing Five) simply uses the information returned from the *svPixelAddr* function to set the window bank and update the pixel. The function appears a little more convoluted due to the logical operations supported. To set the logical operation, use the function *svSetOp* before calling the *svPutPixel* function.

Optimization

These example functions were derived from Genus Microprogramming's current graphics tools that support multiple languages and compilers. Although the routines are fast, the derivation presented here is not necessarily optimized for speed. Optimizations may include streamlining the *svPutPixel* functions themselves so that the *@Entry* and *@Exit* macros are removed, or including separate pixel functions within other primitives such as line or circle functions. The examples are meant to illustrate the Super VGA programming process itself, and modular programming techniques. As my old physics professor used to eloquently understate, "The rest of the proof is trivial, and is left as an exercise for the student." In other words, why hand it to you when you can have fun trying?

Conclusion

The Super VGA modes are, in my opinion, easier to program than most other video modes for two main reasons:

1. The memory is linear, not interleaved or planar, which simplifies buffer addressing.
2. Pixels are addressed as single bytes, which simplifies pixel addressing, masking, and logical operations.

The only constraining factor is the lack of standardization, which has slowed the adoption of the extended modes. With new standards such as VESA on the horizon, this situation should change soon.

DDJ

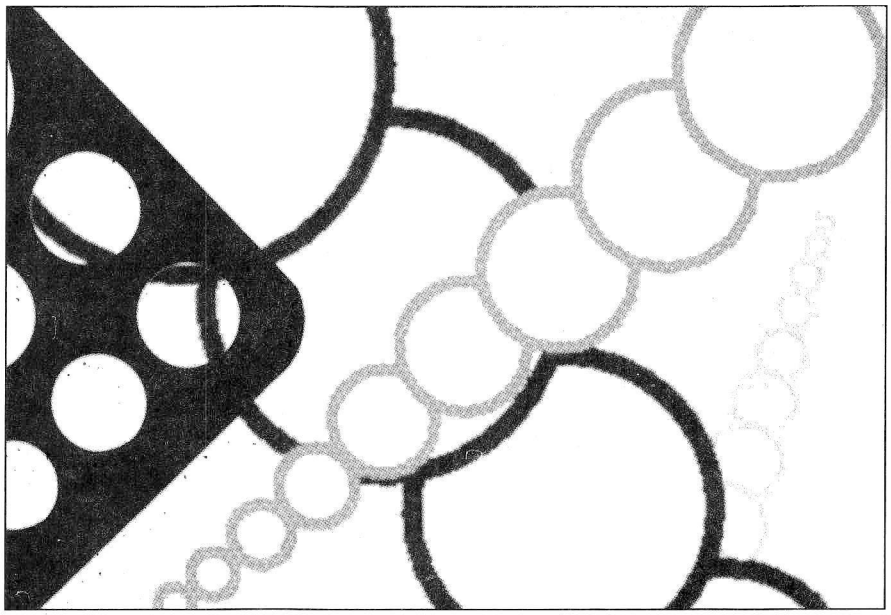
(Listings begin on page 82.)

Vote for your favorite feature/article.
Circle Reader Service **No. 1.**

Circles and the Digital Differential Analyzer

This drawing method belongs in every graphics library

Tim Paterson



Drawing circles has long been a topic of discussion in the pages of *DDJ*. Earlier this year, Robert Zigon ("Parametric Circles," *DDJ*, January 1990) presented a mathematical analysis in which he simplified the brute force approach that would normally require either lots of sines and cosines, or lots of squares and square roots.

Unfortunately, Mr. Zigon got on the wrong track by switching to the polar coordinate system for most of his analysis. His result was an improvement, but still required four floating-point multiplications per point drawn. His answer also required the calculation of a sine and a cosine once for each circle. Another serious drawback was the fact that his equations computed points at fixed angles, which does not translate well into an unbroken curve of screen pixels.

I was sure there was a better way, so I looked into my filing cabinet and pulled out the folder labeled "Circles." In it I found the May 1983 issue of *DDJ*! Daniel Lee had written "A Fast Circle Routine" and had reached a simpler conclusion than Mr. Zigon's.

Both Messrs. Zigon and Lee were calculating the next point on the circle in relation to the last point. If you've had calculus, this sounds like a classic case of computing "the change in y, given a small change in x." In other words, a derivative is called for. The basic equations are shown in Figure 1. In other words, as Mr. Lee put it, "the change in y, given a small change in x, is simply the negative of the ratio of x to y."

The obvious way to use this fact is to compute point n+1 from point n as follows:

$$x_{n+1} = x_n + 1$$
$$y_{n+1} = y_n - x/y$$

As written, these would appear to need to be floating-point operations. However, fixed-point arithmetic would work as well. For example, x and y could be stored as 32-bit numbers, with the bi-

Tim is the original author of MS-DOS, Version 1.x, which he wrote in 1980-82 while employed by Seattle Computer Products and Microsoft. He was also founder of Falcon Technology, which became part of Phoenix Technologies, the ROM BIOS maker. Tim can be reached c/o DDJ.

(continued from page 30)
nary point in the middle: A 16-bit integer part and a 16-bit fraction part. Only the high 16 bits would be used for plotting points. The low bits are needed to keep the circle accurate. Mr. Lee's version wasn't quite this simple, as he scaled *x* and *y* by 1000, but it's the thought that counts.

In order to plot an unbroken curve, two consecutive points cannot be further apart than one pixel in either the *x* or *y* direction. In the equations above, this is always true for *x*, of course, but is true for *y* only as long as $x <= y$. This means the technique can be used on only one octant (one eighth) of the circle at a time. However, circles being so symmetrical, computing one octant is all that's needed to plot the whole thing. For each point computed in one octant, eight points are plotted: Four in all combinations of positive and negative coordinates, and the same four again but with the *x* and *y* values interchanged.

The Digital Differential Analyzer

Mr. Lee's approach still requires a division operation (albeit integer division), and I was sure there was a better way. I found in my folder some handwritten notes on the algorithm used by the run-time library of the Microsoft Basic compiler — circa 1981, when I was part of the team porting it from the 8080 to the 8086. This algorithm uses unscaled integers only, with no operations more difficult than 16-bit adding and shifting. I have since heard this technique called the "digital differential analyzer," or DDA. The DDA is the standard method for drawing both straight lines and circles for every graphics library.

Let's start trying to understand the DDA by examining the case of straight lines. Suppose we have two points that we are drawing a line between, (x_1, y_1) and (x_2, y_2). First we compute two constants, *Dx* and *Dy*:

$$Dx = x_2 - x_1$$
$$Dy = y_2 - y_1$$

Dx and *Dy* are simply how far we have to go in each direction to get from the first point to the second. If we were going to use an approach such as Mr. Lee's, we could compute points on the line like this:

$$x_{n+1} = x_n + 1$$
$$y_{n+1} = y_n + Dy/Dx$$

But *Dy/Dx* is not an integer (usually), so at least fixed-point arithmetic using scaled integers would be required.

The DDA is the standard method for drawing both straight lines and circles for every graphics library

So far, we've been looking at this in fairly "analog" (as opposed to "digital") terms. Yes, we've been incrementing *x* by whole pixels, but *y* has needed to take on noninteger values. The actual drawing operations, however, are interested only in whole numbers for pixel coordinates. Let's change our thinking so that *y* stays a whole number, and we'll keep another variable that will tell us when to increment it.

Basically *y* should get incremented when we've added *Dy/Dx* enough times to equal one more. This is the same as adding in one more *Dy* for each *x* pixel until the total reaches *Dx*. The C code in Example 1 shows the calculation of one additional point on the line. That code would appear in a loop repeated to draw the whole line.

Looking more closely at Example 1, *sum* would have been initialized to *−Dx/2* (or better still, *−Dx >> 1*). For each pixel in the *x* direction, we add *Dy* to *sum*. When *sum* crosses over to being positive, that means we're ready to move one *y* pixel, and we subtract *Dx* from *sum*. *sum* gets *Dy* added for each *x* pixel, *Dx* to subtract for each *y* pixel: It almost amounts to division by repeat subtraction.

But we're really here to look at circles. In the case of lines, we were adding the constant *Dy/Dx* to *y*; as discussed earlier for circles, we need to add *x/y*. That doesn't sound much harder, and, in fact, it's not. Drawing a circle is just as easy as drawing a line. Example 2 has the C code for one pass through the circle's pixel loop, with a few added features for more accuracy.

You should have expected *x* to be added to *sum* each time through the loop; instead we have $2x + 1$. The reason for this is accuracy. Unlike the straight line case, the derivative we're using as the basis for our technique, *y/x*, is continuously varying. As we move from one *x* pixel to the next, which value of *y/x* do we use — the one before or the one after the move? The best answer is to use the value in between. That is, instead of adding *x*, or adding *x*+1, we can add the average: $(2x+1)/2$. But there's no reason to divide by two as long as we do it the same way for the *y* pixel. Note that these lines could also be coded as:

```
sum + = x;
x++;
sum + = x;
```

and

```
sum - = y;
y - -;
sum - = y;
```

By using the value of *x* (or *y*) both before and after incrementing it, we're clearly showing how we want the effect of the midpoint.

The initial conditions for the loop would be *sum* and *x* set to zero, and *y* set to the radius of the circle. But this won't work quite right. Starting *sum* at zero means *y* will be kept less than or equal to its perfect value on the circle for any given *x* coordinate. That is, we'll plot only points that fall exactly on or inside the circle. What we'd like is to plot the nearest point to the circle, whether it be inside or outside it. For the straight line case, we initialized *sum* to *−Dx/2*, which effectively shifted the y coordinate one-half a pixel. But again because the derivative for the circle is varying instead of constant, there is no value that can be used to initialize *sum* to get the same effect.

The solution is to explicitly shift the

$$r^2 = x^2 + y^2, \text{ or}$$
$$y = \sqrt{r^2 - x^2}$$
$$\frac{dy}{dx} = -\frac{x}{\sqrt{r^2 - x^2}} = -\frac{x}{y}$$

Figure 1: *Equation for a circle and ts derivative*

```
sum += Dy;
x++;
if (sum > 0)
{
   sum -= Dx;
   y++;
}
```

Example 1: *The calculation of one additional point on the line*

```
sum += 2*x + 1;
x++;
if (sum > 0)
{
   sum -= 2*y - 1;
   y--;
}
```

Example 2: *C code for one pass through the circle's pixel loop*

(continued from page 32)
y coordinate by half a pixel as we plot the points. That is, instead of having the error from the perfect value for *y* range from 0 to –1 pixel, we'll add one half, and the error will be 1/2 to –1/2 pixel. To do this with integers is simple: Double the radius, so that adding one is equivalent to half a pixel. Then we'll plot only even values of *x*, using the points *x*/2 and (*y*+1)/2.

To add aspect ratio to our circle drawing algorithm requires a little bit of work up front

Example 3 shows a working C function that plots a circle. It calls an additional function *plot8( )*, which actually plots the point generated in all eight octants. Note that all multiplications and divisions by two have been replaced with shift operations << and >>, respectively.

Ellipses and Aspect Ratio

Circles plotted with any of these techniques are round only when the dimensions of the graphics controller, in pixels, correspond to the aspect ratio of the video monitor. Standard video monitors have a 4:3 aspect ratio, so VGA 640 × 480 and super VGA 800 × 600 resolutions produce a "square" pixel (and a round circle). Other typical graph-

```
void circle(int radius)
{
   int x,y,sum;

   x = 0;
   y = radius << 1;
   sum = 0;

   while (x <= y)
   {
      if ( !(x & 1) )
           /* plot if x is even */
        plot8( x >> 1, (y+1) >> 1);
      sum += (x << 1) + 1;
      x++;
      if (sum > 0)
      {
         sum -= (y << 1) - 1;
         y--;
      }
   }
}
```

Example 3: *C function that plots a circle*

ics resolutions (such as CGA, EGA, and Hercules) do not match the 4:3 screen aspect ratio, and will display an ellipse instead of a circle.

Ellipses have their place, too. If we could have arbitrary control over aspect ratio of the circle itself, then we could draw circles or ellipses for every graphics system. And, in fact, any graphics library will have this capability, although it is often expressed in different ways. The CIRCLE statement of Microsoft Basic, for example, requires you to specify the center, the radius, and the aspect ratio. On the other hand, Microsoft Windows and the Microsoft C graphics library want you to specify a "bounding rectangle": You get the biggest ellipse (or circle) that will fit into the box.

To add aspect ratio to our circle drawing algorithm requires a little bit of work up front, plus a single integer multiplication for each point plotted. The simple-minded approach would be to multiply the y coordinate value by the aspect ratio. However, if the aspect ratio is greater than one, this could cause gaps in the arc. In that case, we will multiply the x coordinate by the reciprocal of the aspect ratio. Thus we will always be multiplying by less than one, compressing points on the arc and keeping it unbroken.

Always multiplying by a number less than one has the further advantage of giving us a limit on the range of the number. We can scale the number by up to 16 bits and still store it as a 16-bit number. This is done by *SetAspect( )* in the complete circle drawing program shown in Listing One, page 96. Then as points are plotted, the scaled aspect ratio is multiplied by the x or y coordinate, as appropriate, with the result shifted right 16 bits. Individual points are plotted using the *_setpixel( )* function of the Microsoft C graphics library.

So we now have a complete algorithm for generating circles or ellipses of any aspect ratio. Of course, in a real graphics library, assembly language would be used for maximum speed. And one feature still missing is the ability to draw only an arc, rather than the complete ellipse. The algorithm for a partial arc is no different from what we have so far, but adds a check to see which points we actually want to plot, and which will be discarded.

DDJ

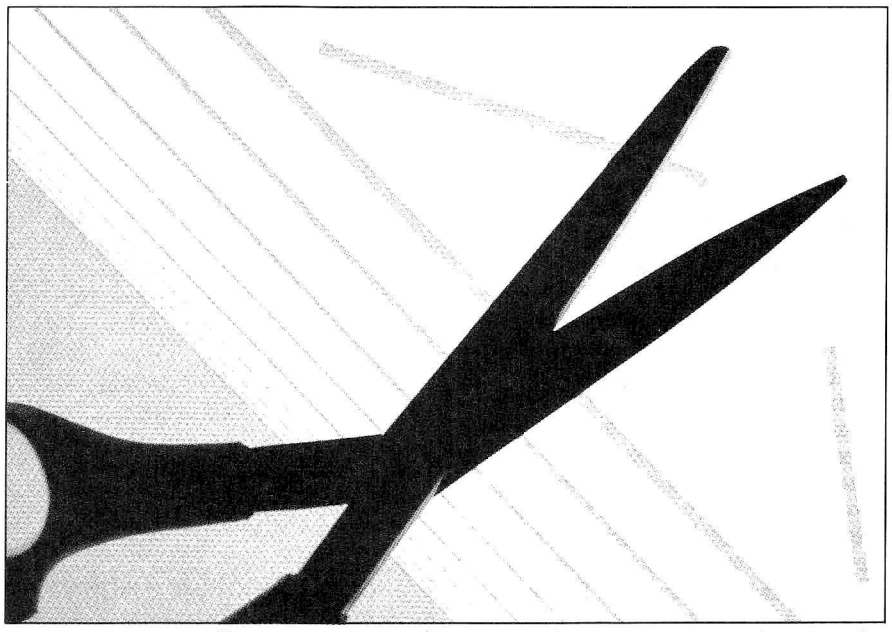
(Listing begins on page 96.)

Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

Improving Line Segment Clipping

More bang for your line clipping buck

Victor J. Duvanenko, W.E. Robbins, Ronald S. Gyurcsik



Over the past few years, windowing environments have become enormously popular. Intuitive graphical interfaces with pop-up menus are now expected of commercial software; successful products such as Apple's Macintosh, X Windows, and Microsoft Windows are thriving *because* they provide these features.

Windowing doesn't come for free, however. Many sophisticated algorithms are behind the creation of the illusion called a window. In this article I'll discuss clipping, one of the key algorithms, by examining Cohen-Sutherland's classic clipping algorithm and then implement — and optimize — it in C. The resulting algorithm will clip about 55 percent more lines per second than Cohen-Sutherland's original algorithm.

As it applies to computer graphics, "clipping" is a process of removing that portion of an object that cannot be seen through a bounding region (that is, a "window"). See Figure 1. Examples of clipping can be found everywhere in everyday life. If you look through the physical window of your house, for instance, you see only a small portion of the outside world. To create the same effect on a computer, decisions must be made as to which portion of the world is to be displayed and which is to be discarded (because the wall that surrounds the physical window conceals that portion of the world). Clipping is the process of removing the portion of the world that cannot be seen.

In computer-aided design (CAD) and other computer graphics applications, very large and complex databases are often created and manipulated. The space shuttle is one such example. When working on a tail section of a project like the space shuttle, engineers don't necessarily want to see the entire space shuttle on the screen. Consequently, the CAD program must be capable of cutting away, or clipping, the undesired sections of the shuttle while displaying only the tail section (appropriately scaled) on the screen.

Because the general topic of clipping is much too broad to cover in a single article, I'll focus on the clipping of line segments that are displayed in a rectangular window that is parallel to the display device axes. This covers most of the present windowing systems that use rectangular windows aligned with the monitor screen. Also, a very large class of physical objects can be drawn with lines.

A Line Segment

To be able to manipulate line segments, you must first understand everything

Victor is a graduate student in electrical and computer engineering with a minor in computer graphics at North Carolina State University. He previously worked as a consultant at Silicon Engineering in Santa Cruz, Calif., and as an IC designer at Intel. He can be reached at 1001 Japonica Court, Knightdale, NC 27545.

(continued from page 36)
about them. How are line segments different from lines? How are lines and line segments described? How are line segments shortened or lengthened? I'll answer these questions in this section.

A "line," in analytic geometry, is a graph of a linear equation Ax + By + C = 0 where A, B are not zero. A line can also be described mathematically as shown in Table 1.[2]

Note that the right side of the two-point form is the slope (m), making the two-point form equivalent to the point-slope form. There are several other ways to describe a line, but they are not as useful to this discussion. For our purposes, a line is a set of all points with coordinates (x,y) that satisfy a line equation (see Figure 2). A line extends infinitely and contains an infinite number of points. This infinity stuff is very troublesome, however. To draw an infinite number of points in a line would take infinitely long, no matter how quickly each point is drawn. Also, there are not many physical objects that contain lines of infinite length. In fact, any such object would have to be infinitely large.

A finite portion of a line is much more useful when dealing with realistic physical objects. Somehow, the boundaries of this finite portion, or segment, of a line must be specified. One convenient way to specify these boundaries is by using two points to indicate the end points of that line segment, and then by using the two-point form line equation. Therefore, a line segment is a set of all points on a line between the end points of that line segment (see Figure 3). There are other ways to define a line segment that are convenient to other applications.

Clipping shortens the line segment. To accomplish clipping, one or both end points of the line segment must be modified. New end points must then be calculated. These new end points must satisfy the line equation in that they must lie on the original line and be within the boundaries of the original line segment (again, see Figure 3). For example, if as in Figure 4 a line segment intersects X_RIGHT (the right window boundary), only the section P1-A should be displayed. The section A-P0 should be discarded. To accomplish this, point A must replace point P0 as the end point of the line segment. Somehow, the X and Y coordinates of point A must be computed.

The X-coordinate of the intersection point must be X_RIGHT, since the P0-P1 line segment intersects the right side of the window (whose line equation is x = X_RIGHT). Only the Y-coordinate is left to be determined. This can be done by solving the two-point form line equation for Y, and substituting X_RIGHT for X in that equation. The resulting equation (shown in Figure 4) is manageable and can be computed quickly. The result is a new end point of a shorter, clipped, line segment. This simple method can be extended to clip lines to the four boundaries of a rectangular window.

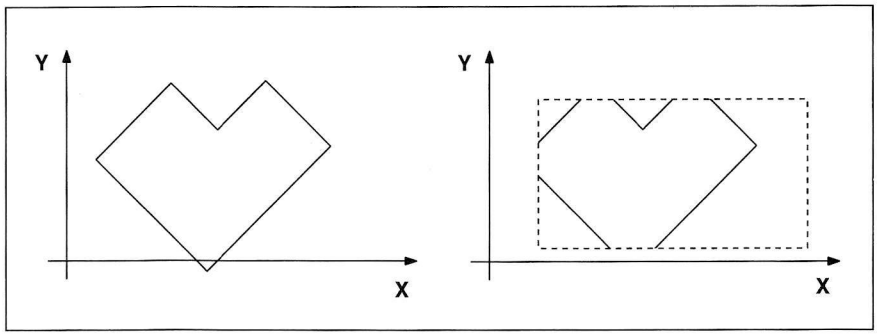


Figure 1: *Clipping is the process of removing portions of an object that cannot be seen through a bounding region*

Form	Math Description
Slope y–intercept form:	$y = mx + b$
Point-slope form:	$y - y1 = m * (x - x1)$
Two-point form:	$(y - y1)/(x - x1) = (y2 - y1) /(x2 - x1)$

Table 1: *Mathematically describing a line*

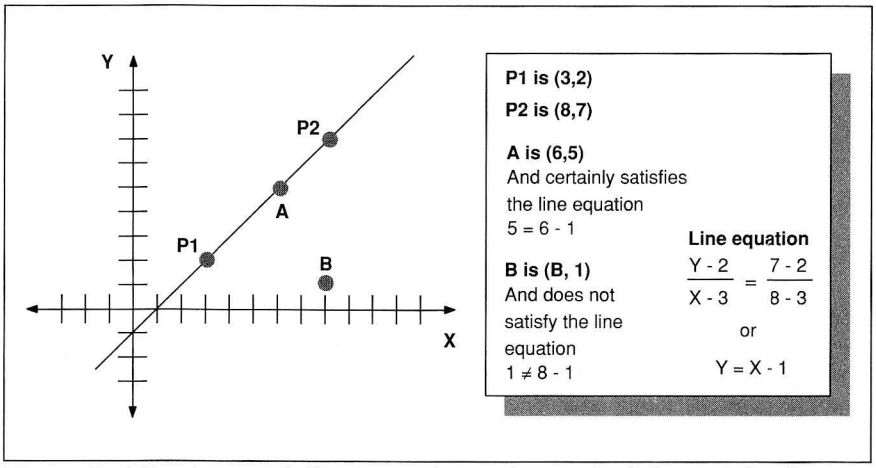


Figure 2: *A line is a set of all points with coordinates (x,y) that satisfy a line equation*

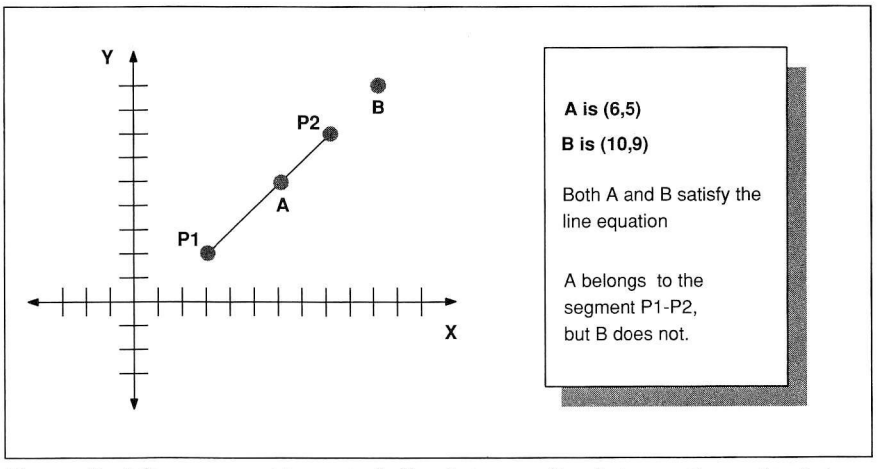


Figure 3: *A line segment is a set of all points on a line between the end points*

(continued from page 38)

A straightforward way to perform line clipping is to find an intersection between the line to be clipped and every boundary of the window and to make sure that the intersection points are within the window limits. If the intersection points are all outside the window limits, then the line can be thrown away. In other words, you must choose the new end points very carefully — between the "best" intersection points and the end points of the original line. This method would require an intersection calculation for all four window boundaries. But is this amount of computation really necessary? Couldn't conclusions be made as each intersection point (up to four of them) is found, to speed up the process? That is precisely what the Cohen-Sutherland algorithm does. The algorithm decides the fate of the line after an intersection with a single window boundary has been determined; it does this up to four times.

The Cohen-Sutherland Algorithm (2D)

The Cohen-Sutherland line clipping algorithm, introduced in 1968, is powerful, simple, and compact. The first adaptation of the algorithm, implemented in C, is shown in Listing One (page 98). This version corresponds to the implementations that are found in literature (see endnotes 3, 4, 7, and 9). An "improved" version of the algorithm is shown in Listing Two (page 98). I will thoroughly explain the code shortly but, first, the method.

Cohen and Sutherland thought of the window boundaries as lines, not line segments. If these window boundaries were extended (lines are of infinite length), the world would be partitioned into nine regions that can be thought of as being within, above, below, to the right, or to the left of the window (see Figure 5). Each region can then be given a unique identifier, an *outcode/number*. This identifier would be more useful if there were some logic behind it, especially if it allowed for quick rejection of line segments that are completely outside the window, or for quick acceptance of line segments that are completely within the window. Cohen and Sutherland suggested the following scheme:

- Bit 0 of the identifier should indicate whether the end point is above the window.
- Bit 1, whether the end point is below the window.
- Bit 2, whether the end point is to the right of the window.
- Bit 3, whether the end point is to the left of the window.

This scheme leads to the region encoding shown in Figure 6. Note that Bit 0 is on the left, so the region encoded with 1010 has bits 0 and 2 set to indicate that it is above and to the right of the window. Each region has now been identified uniquely. For the rest, keep Listing One handy while I lead you through every detail of the code.

The algorithm modifications resulted in a 35.3 percent speed improvement for this sample of 1000 lines. This results in an incredible 54.6 percent more lines clipped per second!

Procedure *clip_2d_cs* is the main procedure of the algorithm. It is called to perform the actual clipping (read the comments for details about the calling conventions). Note the two integer variables, *outcode0* and *outcode1*. These will be assigned to each end point, and will indicate which region an end point lies in.

The algorithm consists of a *while* loop, which is executed up to four times. The first step in the algorithm is to identify in which of the nine regions an end point of the line segment lies and to assign an appropriate outcode to it. For example, if an end point lies in a region that is above and to the left of the window, an *outcode 1001* is assigned to it. This is exactly what the procedure *outcodes* does. Both end points of the line segment get outcodes assigned to them.

Two quick decisions can now be made about the line based on the outcodes. First, a line can be trivially rejected in four cases: If both end points are above the window; if both end points are below the window; if both end points are to the left of the window; and if both end points are to the right of the window. In these cases the line should not be drawn at all, as it lies completely outside the window. An easy and quick way to check for these four conditions is to perform the logical AND of the end point outcodes (see procedure *reject_check*). For example, if both end points are above the window, both outcodes will have Bit 0 set to a 1. A logical AND will produce a 1 in Bit 0, and the line is rejected.

The second decision that can be made quickly is the trivial acceptance if both end points are within the window. In this case, the outcodes of both end points will be 0000. The procedure *accept_check* checks both outcodes and accepts the line if both outcodes are 0000.

At this point the trivial cases have been taken care of, and the more complicated cases must be dealt with. Cohen and Sutherland took the divide-and-conquer approach to the problem. A series of simple adjustments, up to four, is made, and the line is checked for trivial rejection or acceptance after each adjustment. The simple adjustments are exactly like the one demonstrated in Figure 4. Basically, a line is shortened with respect to a single window bound-

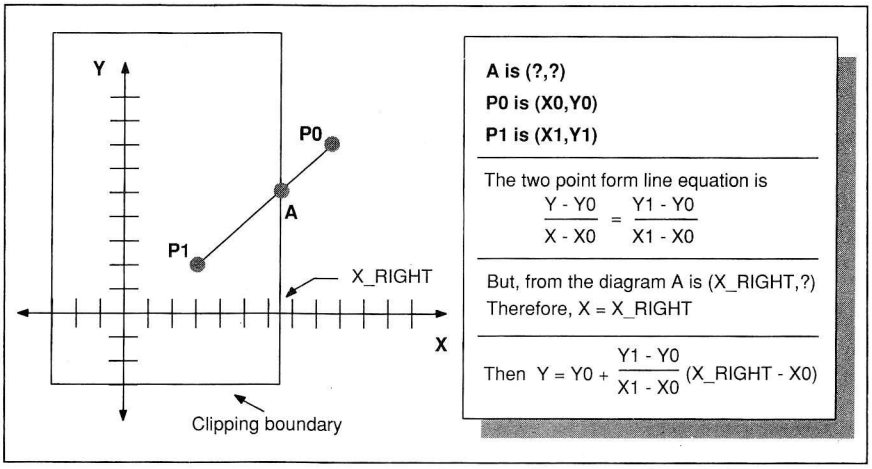


Figure 4: *Computing the X and Y coordinates of point A. The result is a new end point of a shorter, clipped, line segment. This simple method can be extended to clip lines to the four boundaries of a rectangular window.*

(continued from page 40)
ary during each iteration. After up to four iterations, because there are four window boundaries, the line segment has been clipped.

The best way to explain the algorithm is to walk through an example. In Figure 7 the progress of the clipping algorithm is shown one iteration at a time. Line segment end points are first encoded with outcodes. This line segment can't be trivially accepted because both end points are not within the window boundaries (both outcodes are not 0000). This line segment can't be trivially rejected either, because the logical AND of the end point outcodes is 0000. The algorithm then starts operating on P0 end point, because this end point is outside the window boundaries: Its outcode is not 0000. Because Bit 0 is set in the outcode of P0, P0 must be above the top window boundary. P0 can be moved to the intersection of the line segment with the line drawn through the top window boundary. This point will still be on the line segment and will be a bit closer to the window. This action eliminated the section of the line segment that was above the top window boundary and, therefore, could not possibly be seen through the window. The Y coordinate of the new end point is already known: Y_TOP (the top boundary of the window). The only calculation that remains is to compute the X coordinate. This is done by the method similar to that shown in Figure 4, only solving the equation for X, because it — not Y — is the unknown. The code for this is in the *clip_2D_cs* inside the *if (outcode0 & 1)* statement.

Next, because the end points have been moved, they are given new outcodes. Then the new line segment is checked once again for trivial rejection or acceptance. The outcode of P0 shows that P0 is to the right of the window. Therefore, P0 can be moved to the intersection of the right window boundary and the line segment. This action will cause the section of the line segment that is to the right of the window to be discarded (shown in step 2). The X coordinate of this intersection is known, but not the Y, so the algorithm calculates the Y coordinate.

Turbocharging the Algorithm

Now that you understand the algorithm, I'll introduce several modifications to boost the algorithm performance. These modifications are shown in Listing Two. Benchmarks are summarized in the next section.

The first modification is to calculate the slope and the inverse slope only once. This modification comes from a simple idea — the slope of a line before clipping should be equal to the slope of a line after it has been clipped.

The implementation is just as simple as the concept itself. After the first trivial rejection or acceptance has been completed, and failed, the calculation of the slope is inevitable. For half of the cases the slope must be calculated; for the other half the inverse of the slope must be calculated. That's right, if a line can be broken up, or subdivided, at the top or the bottom of the window, then the Y coordinate of the intersection is known and X must be computed. Therefore, in this case the inverse of the slope must be computed. If, however, a line can be subdivided at the left or right of the window, the slope must be calculated.

CAD programs must be capable of cutting away, or clipping, the sections of objects that cannot be seen through a window

If you look at the code in Listing One, you will notice that all computations of **y0* or **x0* involve very similar calculations. In fact, all of them compute *(*x1 – *x0)* and *(*y1 – *y0)*. Realizing that the result of these two computations will be used only in the slope or inverse slope calculations, which do not change, leads to the conclusion that these values need be computed only once. That is what you see in Listing Two. Two variables have been added, *dx* and *dy*, and these are computed only once.

A perfectly reasonable question to ask at this point is, "If we calculated the slope, can the inverse be quickly determined by dividing a one by the slope?" Ah, but there is a problem with this method. Remember the trouble with division by zero — it leads to an answer that computers just can't deal with. In fact, Hearn and Baker[4] missed this point in their implementation of the Cohen-Sutherland algorithm. Their implementation will not work for vertical lines. I hope this serves as sufficient warning for programmers everywhere — check even the most trivial of cases!

The Cohen-Sutherland algorithm care-

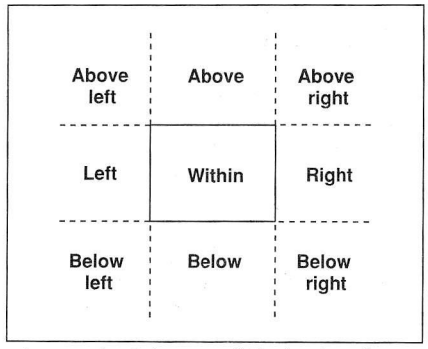


Figure 5: *Cohen and Sutherland thought of the window boundaries as lines, which when extended partition the world into nine regions that can be thought of as being within, above, below, to the right, or to the left of the window.*

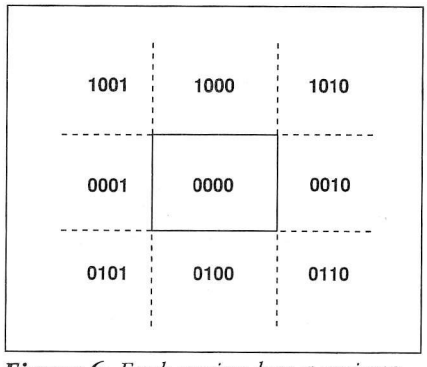


Figure 6: *Each region has a unique identifier, an outcode/number.*

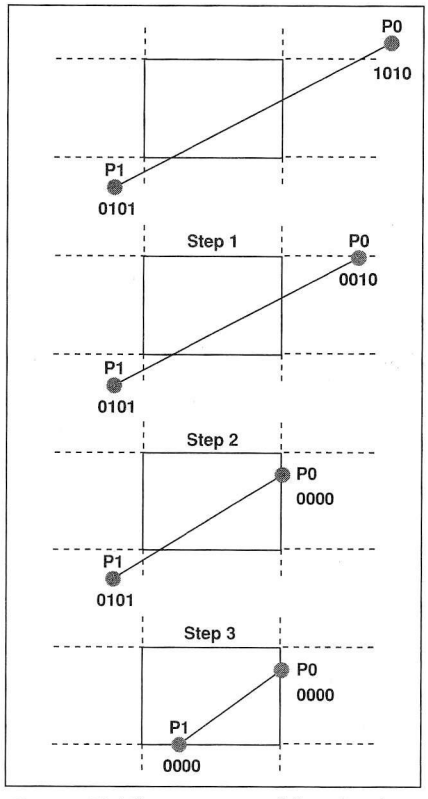


Figure 7: *The progress of the clipping algorithm, one iteration at a time.*

(continued from page 42)
fully avoids division by zero by calculating the slope inside the *if* statements. For example, if a line needs to be subdivided at the top of the window, that line cannot be horizontal. If it were, it would have been either trivially rejected or accepted, or it would be impossible for its outcodes to require subdivision at the top of the window. Also, wouldn't it be a bit difficult to subdivide a horizontal line at the top or the bottom of the window to begin with? The line is parallel to the top and bottom boundary (there is no intersection point).

The second speed improvement comes from several sources (see references 6, 3, and 7). Exercise 4.2 in *Fundamentals of Interactive Computer Graphics* contains a suggestion to encode only P0, the modified end point. This modification has the most dramatic speed improvement on the algorithm. It reduces the number of floating-point comparisons almost in half. Bravo!

This concept can be taken one step further by optimization fanatics like myself. After a line has been subdivided once, only 3 bits are needed for the outcodes. So, only three of the four boundary comparisons are needed. For example, if a line has just been subdivided at the top of the window, there is no need to compare that end point with the top of the window again. You can be assured that the new end point will not be above the window, because we just discarded that portion of the line. Therefore, the comparison with the top window boundary can be eliminated. This reduces the number of comparisons a bit more, but makes the code not quite as elegant as it once was. Well, performance versus elegance has its trade-offs. . . .

The next improvement can be made by noticing another physical property. If a line is being clipped at the top boundary, it is physically impossible for the new top end point to be below the bottom boundary. In other words, the end point that was above the window can move only to one of the three middle regions (0001, 0000, 0010). It will never move below the window, so why check it? Therefore, we can remove yet another floating-point comparison. Now, if the line is being subdivided at the top or the bottom boundary, we need to check against and encode only the right and left boundary. And, if the line is being subdivided at the right or left boundary, we need to check against and encode only the top and bottom boundary.

Now, the number of comparisons, after the initial encoding, has been cut in half. Not bad. But is it possible to do more? Yes, of course. The last improvement is even more dramatic than any of the previous ones. This speed improvement comes from the physical properties described earlier as well. If you look at Figure 6 once again, you will notice that only 2 out of the possible 4 bits are ever set for any region. Why, then, should we do four comparisons? Couldn't we stop after the first 2 bits have been set, and not do the other 2 bits? Yes, we could definitely do that. For example, if we found that an end point is located above the window, we no longer have to check the region below the window. The same holds for left and right regions. This is the reasoning behind all of those mysterious *if/else* statements in the OUTCODES macro and in the body of the main loop. In the best case, only two floating-point comparisons are made. In the worst, still four are needed? The number of comparisons drops dramatically, especially for the trivial rejection case, from eight to four. Note, that trivial acceptance doesn't benefit at all. Newmann and Sproull[6] used this technique in their implementation of the OUTCODES subroutine.

As promised, I pulled all of the C tricks from my bag of tricks. I converted the *outcodes* and *swap_ pts* procedures to OUTCODES and SWAP_PTS macros. This eliminated stack data movement, resulting in further speed improvement. Therefore, the final routine does no procedure calls, and no parameter passing, at all. I had to add a couple of temporary variables that swap procedure used, however, to store the values during swapping.

There are several other small optimization steps that can be taken even further. You could calculate the slope (*dy/dx*) and then set a flag to indicate that the slope has already been computed. Then, if the need arises to use the slope in another *if* statement, that calculation can be avoided. This yields very small performance improvement. It would be beneficial for hardware implementations of the algorithm, however.

Benchmarks

I used the Microsoft C, Version 5.0, compiler and set all performance enhancing switches that I could find, *–Ox –FPi87*. I set the *–FPi87* switch, because I'm fortunate enough to have an 80287 math coprocessor in my 8 MHz, 1 wait state PC/AT. I also used the *–W2* switch, requesting the compiler to find any stupid mistakes that I often make.

To compare the algorithms against one another I wrote a fairly involved, and long, program that will be available through the *DDJ* Forum or on CompuServe. For details see info at the end of this article. This program allows a large database of lines, specified by two end points in three dimensions, to be loaded from a disk file. Then, the user is able to choose from a menu of several line clipping algorithms. The program takes a time stamp before the clipping algorithm is run on the database and when the clipping has been completed. The time difference is then displayed on the screen. The user can run the chosen clipping algorithm as many times as they wish for improved timing accuracy. The window size can be easily modified, and the database can be rotated, scaled, and translated for added flexibility.

To generate the database of line end points, I wrote a short program, shown in Listing Three (page 100), that generates random end points in three dimensions. Yes, I realize that random data is not exactly the best way of benchmarking an algorithm, but I haven't seen a fair benchmark for line clipping yet. The most fair test, in my opinion, would be to generate a tight, paralleled-piped-shaped grid of interconnected points in three dimensions.

The benchmarks in Table 2 were done with the window dimensions set at $5 < X < 630$, $3 < Y < 300$. The values of the line end points varied from $-16K < (X, Y) < 16K$. One thousand lines were clipped 100 times. The resulting clipped lines were drawn in one case and not in the other, giving a true value of the line clipping overhead in the latter case.

The algorithm modifications resulted in a 35.3 percent speed improvement for this sample of 1000 lines. This results in an incredible 54.6 percent more lines clipped per second than the first implementation of the Cohen-Sutherland algorithm!

Additional improvements can be developed to increase the performance further.[10] Presently, a record of about 80 percent more lines clipped has been achieved. The number of comparisons

Algorithm	Draw (in sec.)	Not Draw (in sec.)	Clips/sec.
Cohen-Sutherland	104	85	1,176
Cohen-Sutherland-Duvanenko	74	55	1,818

Table 2: *Line clipping benchmark results*

is the biggest time consumer in the Cohen-Sutherland algorithm.[8] If you think of some other improvements, let me know. I'd love to hear about your results and see the code.

Notes

1. *Concise Encyclopedia of Science and Technology* (New York: McGraw-Hill, 1984).
2. *CRC Standard Mathematical Tables*, 26th Edition (Boca Raton, Fla. CRC Press, 1982).
3. J.D. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics* (Reading, Mass.: Addison-Wesley, 1984).
4. D. Hearn and M.P. Baker, *Computer Graphics* (Englewood Cliffs, N. J.: Prentice-Hall, 1986).
5. S. Harrington, *Computer Graphics: A Programming Approach* (New York: McGraw-Hill, 1987).
6. W.M. Newmann and R.F. Sproull, *Principles of Interactive Computer Graphics*, 2nd ed. (New York: McGraw-Hill, 1979), 121-125.
7. J.R. Rankin, *Computer Graphics Software Construction* Prentice Hall of Australia, 1989), 188 – 194.
8. T.M. Nicholl, D.T. Lee, R.A. Nicholl, "An Efficient New Algorithm for 2-D Line Clipping: Its Development and Analysis," SIGGRAPH 1987 Proceedings, *Computer Graphics*, 21:4, (July 1987).
9. M.A. White and R.J. Reppert, "Clipping and Filling Polygons," *Computer Language*, 3:5, (1986).
10. V.J. Duvanenko, "Point and Line Clipping: Algorithms and Architectures," M.S. Thesis, North Carolina State University, forthcoming, (Dec., 1990).

Source Code Availability

As a service to our readers, all source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Drive, Redwood City, CA 94063, or call 800-356-2002 (inside Calif.) or 800-533-4372 (outside Calif.). Specify issue number and disk format. Code is also available through the DDJ Forum on CompuServe (type GO DDJ).

DDJ

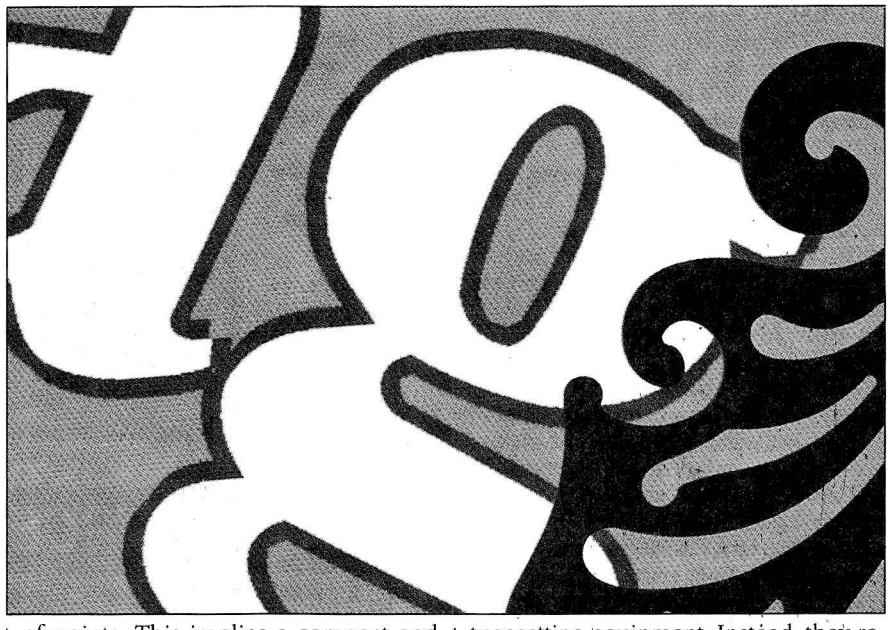
(Listings begin on page 98.)

Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

Drawing Character Shapes with Bézier Curves

A modern approach to an old problem

Todd King



Text is the most common medium of exchanging ideas, theories, and concepts. As we all learned in school, the building blocks of printed text are typographic characters. A complete set of characters of a certain design or style is known as a "font." Currently, there exist more than 10,000 different font designs for the Roman alphabet.

Until the last few years, most computer systems did not devote resources to presenting text in any but the most primitive form: either dot-matrix or stroked characters. With increased computing power, larger memory, and higher resolution displays and printers, personal computer systems can now handle typographic characters with greater fidelity than before.

In this article, we'll take a look at one of the most common ways for representing and rendering typographic character shapes on a computer — the Bézier curve.

Bézier Curves

A Bézier curve is a parametric curve that is specified using a small number of points. This implies a compact and efficient means for storing the definition of a character shape. Bézier curves have other properties that make them particularly useful for representing the shapes found in typographic characters (see next section).

The Bézier curve was originally developed in the early 1970s by the French mathematician Pierre Bézier for Renault (the automaker), as a way to describe the three-dimensional shapes and surfaces of cars. In the two-dimensional world, Bézier curves were not used until recently by manufacturers of digital typesetting equipment. Instead, they relied on straight lines, circles, and arcs to represent typographic shapes — mostly because these were well-understood and easy to compute efficiently.

Adobe, in its implementation of the Postscript page description language, was the first mainstream manufacturer to use Bézier curves. Since then, other manufacturers have followed suit.

The equation that defines a general Bézier curve is a polynomial. This polynomial can be of any order or degree, but as the order increases, so does the computational requirement. Rather than

Todd is a programmer/analyst with the Institute of Geophysics and Planetary Physics at UCLA where he works on various software projects. He is writing a book, Dynamic Data Structures, *which will be published by Academic Press, late in 1990. Todd can be reached at 1104 N. Orchard, Burbank, CA 91506*

(continued from page 46)
try to model a complex shape with a higher-degree polynomial, it is more practical to construct a composite curve that is built by joining together several simpler Bézier curves. These simpler curves are of degree three — the cubic Bézier form.

With this form, a Bézier curve segment is specified by four points, called "control points." Two of these points define the positions of the curve's end points. The other two points control the shape of the curve. Figure 1 shows the important features of a cubic Bézier curve (hereafter referred to simply as Bézier curve).

Representing Characters

A Bézier curve has several properties that make it well suited to representing typographic characters, the first being that the defined shape is both continuous and smooth.

Second, the curve is tangent to the line formed by the end point and the nearest control point. This is useful in creating smooth curves that consist of one or more curve segments joined together. The reason is that if the two curve segments are joined properly then the resulting curve will also be continuous. This occurs if the adjacent curve segments share a common end point, and if the nearest control points for each curve, as well as the common end point, are all collinear.

Third, because the curves are defined by parametric equations, the fonts, which are built up from the curves can be scaled and sized at the time of display. This is done by multiplying the results derived from the equations by some scaling factor.

Bézier curves are also well behaved under other kinds of mathematical transformations, such as rotation and shearing. This was a problem with the line/arc representation used by many digital type manufacturers: The shapes were well behaved for scaling and translation, but not for shearing. (A sheared circle does not remain a circle but becomes an ellipse.)

In addition, Bézier curves have the property of being bounded by the control points, which define it. No portion of the curve extends beyond the edges of the box formed by drawing lines between adjacent control points. If you collapse the box so that all the control points are collinear, then the Bézier curve defines a straight line. This means that Bézier curves can be used to represent both the straight and the curved features of a character.

Finally, Bézier curves can be computed relatively efficiently using a method known as the "deCasteljau" algorithm. While not as fast as Bresenham or other circle-drawing methods, it is fast enough for modern microprocessors.

The Bézier Equation

The general form of a Bézier curve segment is:

$$r(t) = \sum_{i=0}^{n} \frac{n!}{i!\,(n-1)!}\, a_i\, t^i\, (1-t)^{n-i}$$

The cubic form of the Bézier curve, for a two-dimensional coordinate space, can be more easily described by a pair of parametric equations, which determine the x and y coordinates of points on the curve.

The x component of a Bézier curve is defined by the formula:

$$x(t) = (1-t)^3x_1 + 3t(1-t)^2x_2 + 3t^2(1-t)x_3 + t^3x_4$$

The formula for y is the same, except y replaces x in the equation above. The variable t is the parameter for the equation and is varied from 0 to 1.

Designing Fonts

When you design a font it is important to draw the character shapes at a size that will ensure that all features of the font can be accurately captured. Also, because it is possible that the font may be enlarged greatly (for example, as part of a billboard), it is best to draw the prototypical font at an equally large scale.

In Adobe's implementation, character shapes are defined on a grid of 1000 × 1000 points, where a point is 1/72 of an inch. An 11-inch page is approximately 825 points long. So, for typical publishing applications this is a reasonable choice.

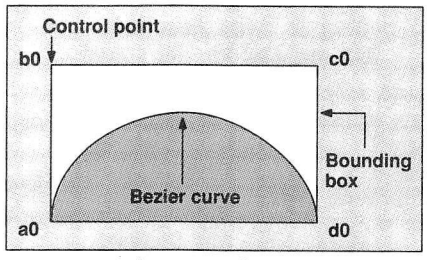


Figure 1: *Elements of a Bézier curve*

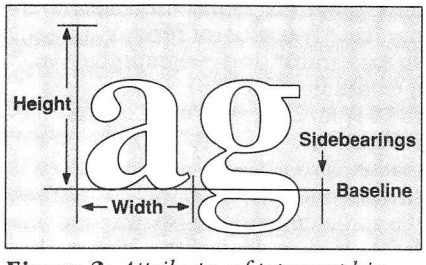


Figure 2: *Attributes of typographic characters*

Characters within a font have certain features or attributes (see Figure 2), which are used in their design. One attribute is the baseline. This is a horizontal line that passes through the font and is used for vertical alignment of a character. All (or nearly all) parts of an "a" remain above its baseline, while the tail of a "g" (also known as its descender) extends below.

Another feature is the sidebearings. This is the amount of space on the left and right sides of a character. The sidebearings can be different for each character in a font, but typically are the same. The sidebearings are included in the total width of the character. In mono-spaced fonts (such as computer line-printer fonts), the character width is the same for all characters. Typographic fonts have proportionally spaced characters, where each character's width depends on its individual design.

You may have heard or read about "hints" buried in the definitions of Postscript fonts (Type 1 Adobe format). These hints describe minor adjustments to the way in which the shape is calculated as the size of the letter becomes very small. This is necessary because the typical display device is a raster device — composed of discrete pixels. As the letter is reduced in size, there may be some distortion of the character due to round-off errors related to the calculations of which pixel a line is in. The "hints" provide information to the rendering system so that the distortion is minimized, and the shape is consistent with the font designer's original intent.

Rendering Bézier Curves

The most straightforward way of rendering a Bézier curve is to use the parametric equations shown earlier. It is not too difficult to express these two equations in C, Pascal, or Fortran, using floating-point values. To calculate a series of points along the curve, just vary the parameter t from 0 to 1 in fractional increments and calculate the corresponding values for x and y.

A faster but less straightforward method of rendering Bézier curves is the deCasteljau algorithm. With this approach, the control box, which surrounds the Bézier curve is divided into two smaller control boxes. These boxes are each successively divided into two smaller boxes, and so on, in a recursive manner, until a specific degree of accuracy is obtained.

At that time, the specific segments

(continued from page 48)
of each of the smallest control boxes are drawn. The net result is that the Bézier curve is approximated by a series of straight-line segments. The first order bounding box is defined by the points *a0, b0, c0, d0.* One of the second order bounding boxes is defined by the points *a0, a1, a2, a3.* The other second order bounding box is defined by the points *a3, b2, c1, d0.* Figure 3 shows how all these points relate to one another. The point *a1* is calculated by finding the midpoint of the line that connects *a0* and *b0.* The other points (*a2, a3, b1, b2,* and *c1*) are calculated in a similar manner.

Bézier curves were originally developed for designing French automobiles

A Sample Program

Listing One, page 102, contains the entire source for a program, which displays a character shape represented by a composite Bézier curve.

To do this, we must first have a function that will draw a Bézier curve segment. In the listing is a function called *draw_bezier1( ).* This function accepts a variable of type BEZIER_BOX, which contains the *x* and *y* coordinates of the four control points for the Bézier curve segment. The function then calculates the points on the Bézier curve from the parametric equations (as discussed earlier), and draws line segments between them.

A second function, called *draw_bezier2( ),* takes the same arguments as *draw_bezier1( ),* but draws the curves using the deCasteljau rendering method, also discussed earlier.

Both the functions *draw_bezier1( )* and *draw_bezier2( )* have been written so that the accuracy at which the Bézier curves are drawn can be adjusted. For *draw_bezier1( ),* the Bézier curve is approximated by drawing straight lines between points, which are separated by a distance, which is on the order of the resolution of the display device.

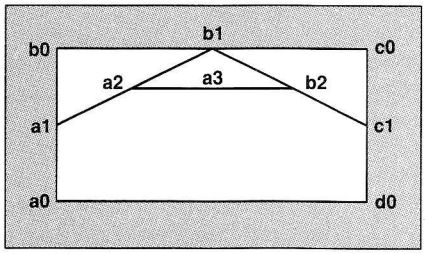


Figure 3: *The control points for the deCasteljau algorithm. Depicted is one level of division of a Bézier bounding box*

This display threshold value is maintained in the variable DPU. To arrive at an appropriate value for DPU, calculate the maximum resolution of the screen and divide that by the size of the prototypical font. Because the DPU depends on the resolution of the display device, the application is somewhat device-dependent.

For *draw_bezier2( ),* the display threshold is determined by the function *accurate( ).* In this implementation, how you determine when the approximation is accurate enough, is to keep a simple count of the number of times the curve has been subdivided. The variable *B_cutoff* contains the number of divisions, which are considered accurate. The variable *B_level* contains the current division count. For screens of resolution of about 600 × 400, a cut-off of 3 is quite good and nearly as accurate as the literal rendering employed in *draw_bezier1( ).* If a cut-off of 4 is used, the resulting characters are indistinguishable between *draw_bezier1( )* and *draw_bezier2( ).*

To make drawing a particular character easier, I included a function called *draw_letter( ).* This function allows you to place the character anywhere on the screen. It also allows you to define the color of the font outline, the fill color, the desired point size to display the character, and the rendering function.

Another argument to *draw_letter( )* is an array of BEZIER_BOX variables. This should contain a series of Bézier curve definitions which, when drawn collectively, will result in the desired character. The array of curve definitions is ended by a sentinel curve definition whose first *x* coordinate is set to BFLAG. The second *x,y* pair of this sentinel curve is the coordinates of a point inside the character's outline. This point is used by the *fill* function.

When you run the program, you will see a series of "a" letterforms printed on the same line in decreasing size. You can choose which rendering method you would like by supplying either the letter "d" (for deCasteljau) or the letter "l" (for literal) as the first command-line argument. You must specify one of these.

Timing

Although my first implementation of drawing fonts with Bézier curves used the literal rendering method, I was

(continued from page 50)

Method	Cut-off	Time (sec)
deCasteljau	3	6
deCasteljau	4	11
literal	*	55

Table 1: *Character rendering timings*

amazed at the difference in speed with which the characters were rendered with the deCasteljau method. At the same time, the quality of the presentation was the same as with the literal method. Table 1 presents some of these timings.

These timings were performed on an AT compatible 286 running at 12.5 MHz with a Hercules graphics display system.

There are a few calls in the source that are unique to Turbo C. These are related to Turbo C's graphics library and area: *closegraph( ), floodfill( ), initdriver( ), lineto( ), moveto( ), setcolor( ), setfillstyle( )*, and *setlinestyle( )*. One call whose arguments may need changing is *initdriver( )*. The last argument in this call is the path to where the device drivers are stored. You may need to change this to match your system's configuration.

Improvements

There is a lot of room for extending this program. The most obvious is that the font has only one character in it — the letter a. This allows for only limited expression. It actually took me a couple of hours to develop the letter (from scratch) to an acceptable form. Some letters in the alphabet should be easier and a few would be harder.

Another improvement would be to design a structure that would contain character-width information for an entire font. I chose not to do this for reasons of simplicity.

Other possible improvements include speeding up the renderer and improving quality at low-resolutions by adding a hinting mechanism.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listing begins on page 102.)

Vote for your favorite feature/article.
Circle Reader Service **No. 4.**

Information Models, Views, and Controllers

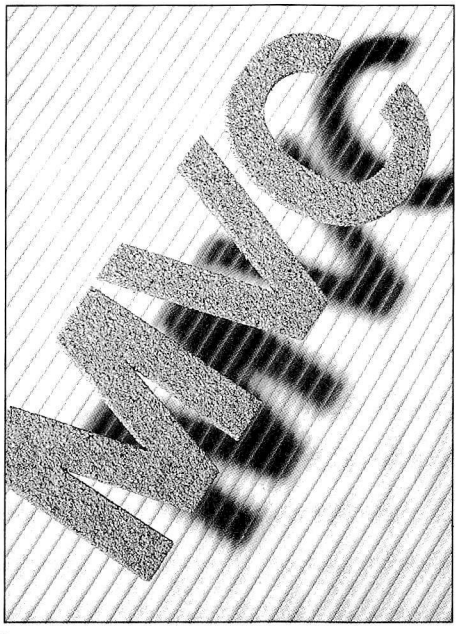
The key to reusability in Smalltalk-80 lies within MVC

Adele Goldberg

We have all experienced the benefits of reuse in every aspect of life. For example, when traffic control signs are reused throughout a city, a state, and even internationally, our ability to navigate streets and highways is helped. Reuse of text editors and picture editors across applications on the same or different hardware systems improves our ability to use those applications. The value of a graphing or diagramming technique increases with the number of ways it can be reused on different data, thereby improving our ability to quickly analyze new kinds of data. Reuse benefits the user as well as the application developer.

In the context of software applications, to reuse a software component generally means keeping it the same, while changing the context or style in which it is used. Text, for example, might stay the same, while the fonts and document layout containing that text might vary; data might stay the same, but it might be presented differently in terms of graphs, pie charts, tables, or animations; or the ways of presenting information might stay the same, while the techniques for making selections and issuing commands might be different.

Adele is the president and CEO of ParcPlace Systems. She can be reached at 1550 Plymouth Street, Mountain View, CA 94043.



One major area of software development that can benefit from reusable software is the design and implementation of applications having graphical user interfaces. As a consequence of the widespread availability of bit-mapped displays, the attention of many programmers shifted from emphasis on implementing models of data, transactions, and/or business functions, to creating appealing user interfaces. Requirements for adhering to the many user interface standards have not decreased the importance of seeking innovative ways to present information on the display screen and to support users in accessing and manipulating that information.

Because reusability of user components is a key element in improving the productivity of software developers, an important question to answer is which software components should be targeted for reuse? This article answers the reuse question through a review of the implementation architecture called "Model-View-Controller" (MVC) that is available to application developers using the Objectworks for Smalltalk-80 program development system developed by ParcPlace Systems. The intent of the article is to demonstrate the advantages of this architecture in developing graphical interactive applications.

MVC

The Model-View-Controller architecture for the Objectworks for Smalltalk-80 system was designed specifically to help programmers create graphical user interfaces through the reuse of software components that represent the presentation and interaction aspects of an application. The use of object-oriented software was essential in motivating the design for MVC and was the basis for its implementation.

One way to improve productivity in software development is to reuse user interface designs across different applications. Another entails leveraging special support for multiple ways of view-

(continued from page 54)
ing and interacting with an information model in accordance with diverse user preferences. For example, some users might want to view their data as tables of numbers, while others might prefer graphs, descriptive text, or animations. These alternatives might be mapped onto a set of software components that can be attached interchangeably to a common information model. The information model is then shared among the projects requiring the different graphical interfaces. The development effort and cost involved in implementing the information model is incurred just once. Many different information models can be presented with similar user interfaces.

Recognizing and exploiting these opportunities for reuse led to the design of two kinds of elements in Objectworks for Smalltalk-80 — a library of presentation and interaction components, and a set of tools supporting a methodology for linking presentation and interaction components to underlying information models. These kinds of elements provide the basic building blocks for the MVC architecture.

"Model" refers to a representation of the application domain as an information model, while "view" is a specification of how aspects of a model are presented to the user. The "controller" is the specification of how the user can communicate or interact with the application in order to request changes in the view or in the underlying model. Consider, for example, a real-time clock that is to appear on the display screen. The information model is an object representing *DateAndTime*, which is implemented in terms of the computer hardware's built-in clock. Suppose *DateAndTime* can report the current time of day and the name of the current day of the current month. On the screen, *DateAndTime* might appear as a digital watch in which the day and month are printed on one text line and the hours:minutes:seconds appear, constantly updated, on a second text line. This text-oriented presentation is the "view" of the model *DateAndTime*.

The ability to reuse software components depends, of course, on the ability of the software system

Another view might be of an analog clock, with a second hand moving around, ticking off seconds. Perhaps the user has direct access to resetting the time information. In the textual view, the user could select the text of hours:minutes:seconds with a pointing device such as a mouse and then use a text editor to modify the visible information. This in turn would modify the underlying *DateAndTime* model, and the change would be reported back to the view that is an analog clock for immediate update. The user's ability to text edit is handled by a "controller."

The design of the MVC roles in Smalltalk-80 came about as a result of a two-stage factoring. First, the design and implementation of the domain-specific aspects (the model) is separated from work on the user interface. Second, the user interface is divided into presentation and interaction aspects.

The ideal scenario for the construction of application software under the MVC paradigm begins by focusing on the design and implementation of the

information model components of an application. (Even here, there are substantial opportunities for reuse from previous projects.) Once the underlying model is completed, the developers reach into a library of reusable user interface components and select (and tailor) various graphical presentations and styles of user interaction appropriate for the target users' preferences. In this way, it is possible to create several implementations quickly, varying the presentation and interaction styles according to user needs. In our experience, it is common for the underlying model and presentation style to stay fixed across a broad set of end users, whereas interaction style (such as typing commands versus command keys, or the selection of icons) varies considerably. By separating the notion of presentation from interaction, this disparity can be accommodated.

Reuse Through MVC

The ability to reuse software components depends, of course, on the ability of the software system or language to describe and maintain identifiable modules. The current wisdom in the software industry is that such a module is a packaging of behavior (procedures) and properties that both describe the module and comprise the data required to implement the behavior. The module is referred to as an object; a system that supports the description, implementation, and testing of objects is said to be object oriented. Such a system must also provide tools for factoring and re-factoring modules in terms of abstract and concrete specifications of behaviors to be represented by modules of various scopes and complexities. The software developer implements an application by creating objects that combine, refine (or specialize), and/or establish information dependencies among existing (that is, reusable) and newly synthesized object types. The MVC approach suggests factoring a system into three kinds of objects. Model objects share a generic ability to inform views and controllers of changes in their state. View objects understand how to draw graphical representations of a Model within an identified area of the display screen and how to update those representations according to information solicited from the Model. Controller objects support a variety of default- and developer-defined ways of handling user events, scheduling access to Views supporting user interaction via input devices, and sending editing commands to Views or Models. Each of these three kinds of objects is defined in terms of a hierarchy of object descriptions, as shown in Figure 1.

Reusing the Expertise of Others

The technical history of MVC is tied to the idea that programmers are not typically trained as graphic artists nor as human-factor engineers. But the knowledge of such experts could be captured, in the form of presentation techniques and interaction frameworks, and made available in a library of reusable software components. The factoring of MVC draws attention to particular kinds of components that should be made available for reuse in order to take advantage of the knowledge of these experts.

Two programming techniques are needed to support such reuse: Composition and refinement. Composition entails combining existing elements to create something new. Objectworks for Smalltalk-80 supports two kinds of composition: Composition via delegation, stating that a newly created object has properties that refer to (instances of) existing classes (an action that requires manipulation of these properties is delegated to the appropriate object); and composition via dependency, stating that an instance of one class depends on the change in behavior of (instances

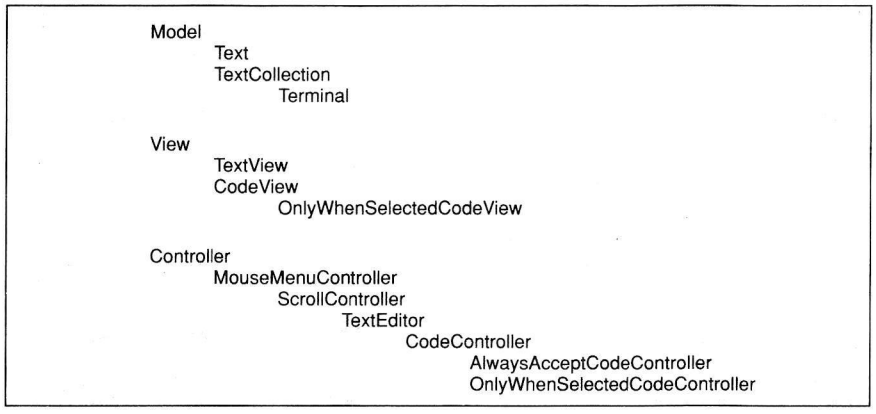


Figure 1: *Hierarchy of Models, Views, and Controllers for Text (each subclass adds a refinement in behavior)*

of) another class.

Additionally, two kinds of refinement are supported: Refinement via inheritance (subclassing), stating that a newly created object acquires the properties, protocol, and implementation of the protocol, of an existing class; and refinement via parametrization, stating that a new object is an instance of an existing class of objects by filling in details about the properties that distinguish this instance from other instances of the class.

Through composition, you can reuse existing models (that correspond to generic objects), existing views, and controllers that define a look and feel for the user interface and existing tools, for example, a text editor or document outline browser.

Through refinement, you can reuse existing application-building frameworks that encompass sets of views and controllers, as well as parametrisizable models. You can also reuse existing parameterizable, tailorable complete applications.

Through composition and refinement, object-oriented MVC design allows the software analyst/designer to focus on creating components that are immediately reusable and that provide for future incorporation of new views, controllers, and/or models.

A Model for Counting

The Smalltalk-80 code in Listing One (page 106) consists of a simple model for maintaining a numeric counter, associated views, and controllers. This model is called "Counter." Using a variety of viewing objects, the current value of the counter is presented on the display screen in five different ways. Two viewing classes are offered in the example: *CounterView* and *BarView*. The same controller is used for all of these views. It specifies that the user selects a menu item, either "Increment" or "Decrement," in order to change the numeric value by an increment or decrement of one. Presentation of a menu that appears when the user presses a mouse button is handled by the controller, an instance of class *CounterController*.

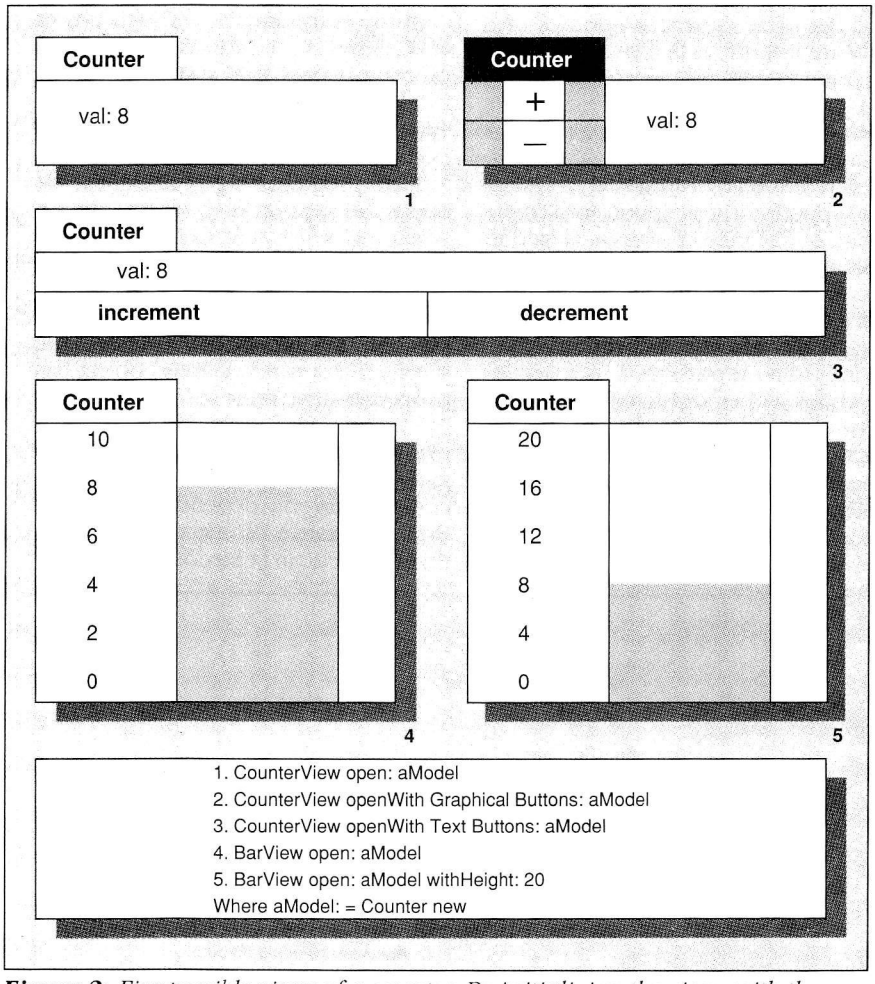


Figure 2: *Five possible views of a counter. By initializing the views with the same model of a counter, all views present the same numeric values, and each view updates whenever the counter value is changed.*

The five examples are shown in Figure 2, and the initialization messages to classes *CounterView* and *BarView* are given in the comment of the code for the classes.

Note that in the Objectworks for Smalltalk-80 system, the library of reusable components includes the classes shown in Table 1, in addition to classes *Model*, *View*, and *Controller*.

The algorithm for displaying a view is the response to the message *displayView* as defined in classes *CounterView* and *BarView*. Note that in Listing One, double quote marks surround comments that are dispersed throughout the code. Formatting is done to aid readability.

The use of object-oriented software was essential in motivating the design for MVC and was the basis for its implementation

By initializing each view with the messages shown in Figure 2 with the same model of a counter (that is, where *aModel : = Counter new*), all the views present the same numeric values and each view updates whenever the counter value is changed.

Advantages of the MVC Approach

Three primary advantages derive from adopting the MVC approach and using it to factor graphical interactive applications. The advantages are: Multiple Viewing, Development Productivity, and Quality. Let's look at each briefly.

Multiple Viewing. Factoring the underlying model from its graphical presentation and interaction allows the programmer to couple the model to several alternative interfaces. Objectworks for Smalltalk-80 provides special support for objects to exchange information about changes. Whenever an object changes, it broadcasts a message that it has changed and what aspect (property) has changed. By combining this support for dependency relationships with the factoring of views and controllers from the underlying model, several views of the same model can be inter-

(continued from page 59)
acted with simultaneously on the same display.

Development Productivity. As noted earlier, it is often possible to create new views and controllers as refinements of existing ones, while retaining the underlying information models. It is also often necessary for the programmer only to implement specific kinds of models that fit into existing user interface designs. Therefore, taking advantage of the reusability of existing MVC components and frameworks can reduce significantly the amount of programming necessary for completing an application development project.

The programmer must be aware of the contents of the available libraries of reusable components and have tools for locating and experimenting with those components. Objectworks for Smalltalk-80, for example, provides the system source code browser with MVC components carefully organized into categories. Each of the program development tools provided with the product are created using the MVC approach. The programmer can inspect the implementation of these tools using the ParcPlace debugger and special MVC inspector.

MVC factoring also facilitates the programmer's ability to respond to new technology for special physical needs or interaction media. For example, this factoring makes it a simpler task to utilize voice output instead of printed text on a display screen. Only the presentation aspects of an MVC-style application would change, not the underlying model.

Quality. Reusing existing components allows those components to become more mature and, therefore, more robust as they are tested in new situations. Reusing existing designs supports the acquisition of the expertise that might not otherwise be available. And improved productivity allows for more experimentation and testing.

The Future

MVC was first explored and tested in earlier versions of the Objectworks for Smalltalk-80 system. It was designed to enable reuse of classes representing views and controllers, with models that were otherwise nongraphical. The current implementation was designed with assumptions about the sophistication of the programmer who is able to reuse components from an extensive library. The desire for uniformity of software architecture encouraged the application of the MVC design concepts to text and picture editors, creating considerable experimentation and discussion about the idea of splitting techniques of selection, scrolling, and zooming from these underlying graphical entities. The original MVC was not designed to deal specifically with underlying models that are themselves graphical in nature. We were primarily trying to support visualization of simulations written in Smalltalk-80, itself designed as a language that supports general-simulation descriptions. Over years of use, we have discovered simplifications and new abstractions that make these performance and implementation issues easier to understand. New improvements will appear in subsequent releases of Objectworks for Smalltalk-80.

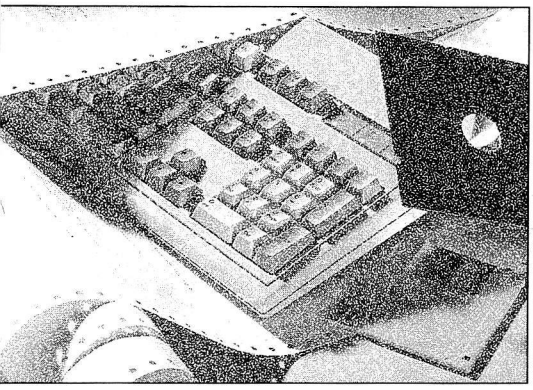
Source Code Availability

As a service to our readers, all source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Drive, Redwood City, CA 94063, or call 800-356-2002 (inside Calif.) or 800-533-4372 (outside Calif.). Specify issue number and disk format. Code is also available through the DDJ Forum on CompuServe (type GO DDJ).

DDJ

(Listing begins on page 106.)

Vote for your favorite feature/article.
Circle Reader Service **No. 5.**



DOS + 386 = 4 Gigabytes!

Directly address 4 gigabytes of memory in DOS from your C or assembly language applications

Al Williams

Ever since Intel introduced the 8088 and 8086, programmers have chafed at the 64K limit imposed by the 8086's segmented architecture. Dealing with data structures greater than 64K has required great feats of legerdemain and been all but impossible in some high-level languages. The 80286 came along, but it still used 64K segments; and though the 286 can address 16 Mbytes of memory, DOS knows only how to deal with the first megabyte. Then the 80386 arrived on the scene. At last, programmers could define segments ranging in size from 1 byte to 4 gigabytes!

Unfortunately, DOS still limits programmers to 1 Mbyte. In this article, I'll show a method for accessing the entire 80386 address space (4 gigabytes) as one flat range of addresses. I'll also provide support for accessing memory from C, controlling the 80286/80386 address lines, allocating extended memory, and adding assembly language to C programs without an assembler. The programs presented all compile under Microsoft C 5.1 with or without the Microsoft assembler, MASM 5.1. Mix's PowerC also compiles these programs.

Addressing Revisited

Recall that the 8086 uses a model of memory addressing known as segmentation to break memory into pieces (or segments). Inside each segment, each particular byte has a unique offset. To address a byte of memory, both its segment and offset must be known. A full address is usually specified as SSSS:OOOO, where SSSS and OOOO represent the segment (or segment selector) and the offset, respectively. The exact interpretation of the segment selector depends on the operating mode of the 386.

In real mode, all segments are exactly 64 Kbytes long. The first segment starts at the bottom of memory, and each consecutive segment starts 16 bytes after the previous segment. Because a 16-bit number (the selector) represents a segment, the address space covers 65,536 × 16 = 1,048,576 bytes (or 1 Mbyte). Memory above 1 Mbyte is normally not accessible in real mode.

One interesting difference between an 8086 and an 80386 (or 80286) in real mode is the handling of addresses at the 1-Mbyte boundary. Generating an address, for example, of FFFF:0011 on an 80386 actually addresses above the 1-Mbyte limit. An 8086 will "wrap-around" so that the address generated is actually the same as 0000:0001. Since some programs may depend on this wrap-around, 80286 and 80386 motherboard designers have added a "gate" for the address line above 1 Mbyte (the A20 line). With the gate turned on, wrap-around doesn't occur. With the gate turned off, as it usually is, addresses appear to wrap around as on an 8086.

Protected mode treats segments differently. In protected mode, a segment selector contains a 13-bit number that indexes into one of two tables known as descriptor tables. One bit in the selector determines which table to use, and 2 bits control segment use (the privilege level, which we won't use), for a total of 16 bits. The descriptor table stores the segment's start address, length, and other pertinent data. Figure 1 shows a segment selector and a partial descriptor table. For our purposes, we need only part of the information in the Global Descriptor Table (GDT).

Each entry in the GDT is 8 bytes long. If the 80386 loaded each entry from memory every time it accessed memory, performance would suffer greatly. To prevent this from happening, the 80386 caches each entry internally whenever the program loads a segment register. In real mode, the processor never changes the cache, because the descriptor tables are not used.

Note that in real mode, any two numbers you put together form a valid address. In protected mode, however, only certain segment selector values are valid. In protected mode, a segment's length determines which offsets are legal. If you try to use a segment improperly or address outside of its range, the 80386 will generate an error. When switching from protected mode to real mode, Intel recommends setting all of the segment registers to selectors that have a 64K limit before switching to real mode. If, however, you disregard the documentation and set the segment registers to selectors with a different limit, the 386 retains that limit during real mode. Set up protected-mode

While Al's programming endeavors range from AI to real-time control software, he specializes in system-level software. Be sure to look for Al's 386 DOS extender in an upcoming DDJ. *Al can be reached via CompuServe (72010, 3574) or at 310 Ivy Glen Court, League City, TX 77573.*

(continued from page 62)
segment registers with a 4 gigabyte limit before returning to real mode.

The Plan

To successfully address the entire memory space from real mode, you must perform the following steps:

1. Disable interrupts, including Non-Maskable Interrupts (NMI)
2. Switch to protected mode
3. Load one or more segment registers with a "big" (4 gigabyte) segment
4. Switch back to real mode
5. Enable interrupts

Once these steps are performed, the segment registers remain affected until a processor reset or until another protected-mode program reloads them. Because real mode does not use segment descriptors, the descriptor cache is never reloaded.

For DOS use, it is desirable to provide routines to:

- Control A20 gating
- Manage allocation of extended memory
- Move data by using the new segmentation scheme and the 32-bit registers

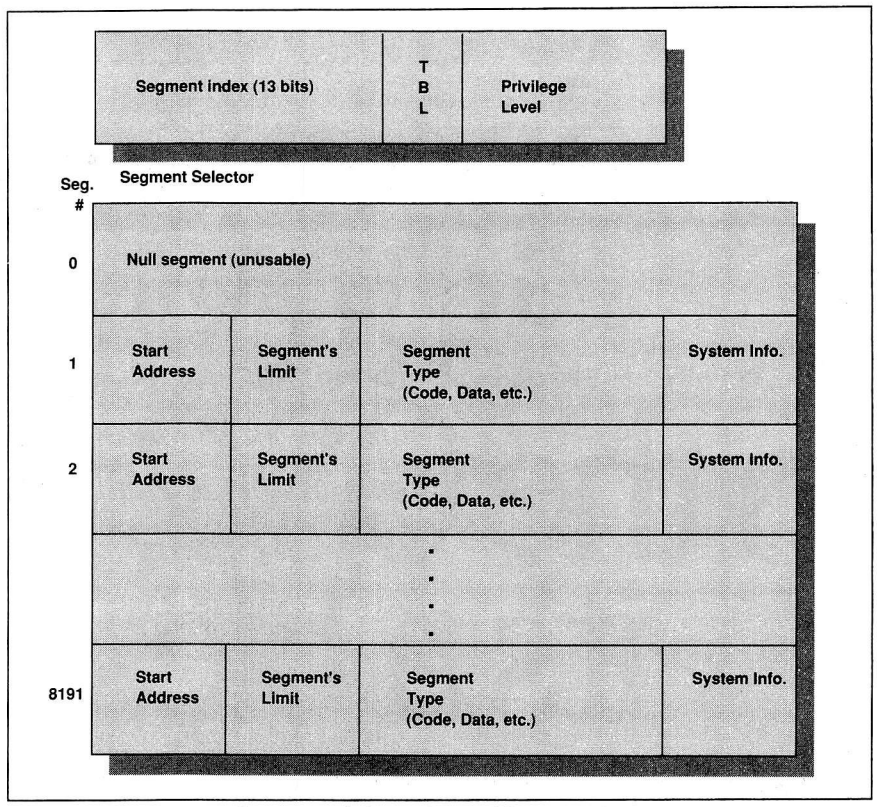


Figure 1: *Segment selector and descriptor table*

(continued from page 64)

- Convert between linear addresses and segmented addresses

Listings One through Five show the SEG4G library that performs these functions.

Since some programs may depend on 8086 wrap-around, 80286 and 80386 motherboard designers have added a "gate" for the address line above 1 Mbyte (the A20 line)

Some Assembly Required

Obviously, to switch modes and perform other 386 magic, we need some assembly language routines. However, not everyone has access to an assembler that generates 80386 protected-mode code. Because of this, you may select one of three different methods to generate the assembly language code. The first method uses Microsoft's assembler (MASM Version 5.1). The second and third methods are for Microsoft C Version 5.1 and Mix's PowerC (see Listing Five, page 112), respectively, and do not require an assembler.

While PowerC provides an *asm( )* function, Microsoft does not. The macro contained in ASMFUNC.H (Listing One, page 110) remedies this absence. This macro allows you to create a character array containing the machine code you want to execute and then call it as a function, complete with arguments and an integer return value.

Before compiling, you must select one of the assembly methods (ASM, DATA, or POWER) at the top of SEG4G.H (Listing Two, page 110). If you pick ASM, you must assemble SEG51.ASM (Listing Four, page 111) separately and link it with SEG4G. Be sure to change the .MODEL directive at the top of SEG51 to match the model you are using for your C programs. In addition, if you use an Intel Inboard 386/PC, set the variable *inboard* to 1.

(continued from page 66)
(Defined near the top of SEG4G.C, Listing Three, page 110.)

Using the SEG4G Library

To force the segment limit on the GS and ES registers to 4 gigabytes, call the *extend_seg( )* routine. This call modifies the registers until the computer is rebooted. If you plan to access extended memory, you must also enable the A20 line by calling the *a20( )* function. The call *a20(1)* turns on A20, and *a20(0)* turns it off again.

The library defines a new data type, the LPTR. This is simply a 32-bit linear address pointer implemented as an unsigned long. For example, the start of the CGA video buffer (B800:0000) is equal to an LPTR of 0xB8000. Two of the supplied functions convert LPTRs to C far pointers and vice versa. Call *linear_to_seg( )* or *seg_to_linear( )*, as appropriate.

While preparing to access memory, you may wish to allocate extended memory. The most common method for allocating extended memory is the "top-down" method. This method temporarily reduces the amount of extended memory reported by the BIOS. For example, if you have 1024K of extended memory and you allocate 24K, other programs calling the BIOS will be told that only 1000K of memory is available. Because extended memory always starts at the same place, the memory is allocated top-down. The only major program that does not use this method is DOS's VDISK (or RAMDRIVE). It uses a peculiar scheme that varies from version to version of DOS. However, because VDISK uses memory from the bottom up, you can control allocation of each so that they never overlap.

SEG4G provides several functions to manage extended memory allocation. Most of these functions are only of interest if you plan to stay resident or run other programs that use extended memory from inside your program. If you don't do either of these things, you can simply check how much extended memory is available and then use it as you see fit. To check the amount of extended memory that is available, call *ext_size( )*, which returns the number of 1K pages that are free.

If you actually need to allocate extended memory, you may use the *ext_alloc( )* and *ext_realloc( )* functions. These functions each take the number of 1K pages desired and return an LPTR to the start of the memory block. If the request cannot be honored, the routines return (LPTR)-1L.

Note that these functions are not like the traditional *malloc( )* functions found in the standard C library. You should not call them repeatedly to allocate small chunks of memory — allocate all of the extended memory you need in one call. This is especially true of programs that are resident. If another program has allocated extended memory after your first call to *ext_alloc( )*, you will be unable to expand your memory allocation.

When you have finished using the extended memory allocated, free it with the *ext_free( )* function. This function frees all of the extended memory allocated in your program. Exercise caution when using *ext_free( )* with other programs that use extended memory. *ext_free(1)* forcibly frees all extended memory allocated since the first call to *ext_alloc( )*. If you have allocated extended memory, you must call *ext_free(1)* before you exit your program. Failure to do so will lock up the computer. Calling *ext_free(0)* attempts to free up the memory, but won't forcibly do so if another program has also allocated extended memory.

Once you have done all of the required setup, you are ready to access memory. The functions *big_read( )*, *big_write( )*, and *big_xfer( )* will read, write, and move blocks of memory, respec-

(continued from page 68)
tively. These functions do not have to operate on extended memory — they work on any linear address.

The use of *big_read( )* and *big_write( )* is straightforward. The *big_xfer( )* function, however, becomes more efficient when you obey certain rules. In particular, performance is best when you move 32-bit words that are aligned on

If another program has allocated extended memory after your first call to ext_alloc( ), *you will be unable to expand your memory allocation*

32-bit boundaries. For example, moving 128 bytes from location 0x42050 to location 0xb8000 is very fast; moving 127 bytes is somewhat less efficient, and moving 128 bytes from location 0x42051 to location 0xb8000 is also somewhat slower. The *big_xfer( )* function tries to optimize transfers by making as many full-word moves as possible. It also attempts to move as much on word boundaries as possible.

Examples

TEST.C (Listing Six, page 112) shows an example program using the SEG4G library. (If you are using VDISK or RAMDRIVE, be sure that you have at least 2K of extended memory not being used by the RAM disk before running this program.) TEST calls *extend_seg( )* to set up the 4 gigabyte segments, enables A20, and then attempts to allocate 1K of extended memory. If successful, it writes a data byte to the entire block and then tries to read it back. Next, the block is expanded to 2K and freed. At this point a loop executes so you can examine memory anywhere in the computer's address range. Figure 2 shows a session with the test program and the RAMDRIVE driver installed. Notice the RAMDRIVE message at the start of extended memory. When you are ready to leave the program, enter a Ctrl-Z.

Listing Seven, page 115, shows BLKTEST.C, an example of using *big_xfer( )*. Because the program writes directly to the screen, you must change the COLOR define to match the type of display your computer has.

Conclusion

The SEG4G library offers a fast, simple method to access the entire 386 memory range from DOS. Even programs that are not running on a 386 can make use of the extended memory allocation, the A20 control routines, and the assembly language interface macro presented here. SEG4G can help implement memory intensive applications such as expanded memory drivers, ram caches, speech/video buffers, and databases.

The extended memory allocation routines allow SEG4G to coexist peacefully with other extended memory-aware programs, but they won't protect it from applications that assume they own all of the extended memory available. In addition, DOS extenders, multitaskers, memory managers, and other software that use protected or virtual 8086 mode may not be compatible with SEG4G.

As with any undocumented feature, this one could vanish at any time. However, it is unlikely that the segment cache scheme used in the 386 will change any time soon. While SEG4G may not be the answer to all of your memory problems, it can provide you with more usable space under DOS, along with some working experience with the 80386's protected mode.

```
C:\SEG4G>SEG4G

1280K of extended memory available
1K of extended memory allocated at   10FC00. 1279K remains.
Data written to extended memory

Data read back OK.
Expanding allocation to 2K
2K of extended memory allocated at   10F800. 1278K remains.
Extended memory freed. 1280K Available.
Enter ^Z to quit.
Address and count? 0x100000 256
MICROSOFT.EMM.CTRL.VERSION.1.00.CONTROL.BLOCK . . . @ . . .
. . . . . .
. . . . . .

Enter ^Z to quit.
Address and count? ^Z

C:\SEG4G>
```

Figure 2: *Typical session using TEST.C with the RAMDRIVE driver installed*

Bibliography

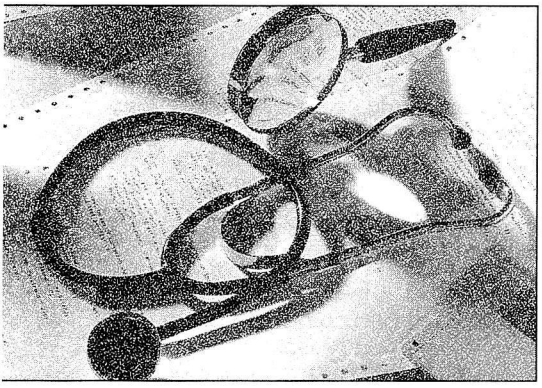
Turley, James L., *Advanced 80386 Programming Techniques, Osborne/McGraw-Hill, Berkeley, Calif., 1988.*

Intel Corporation, 80386 Programmer's Reference Manual, Intel Corp., Santa Clara, Calif., 1986.

DDJ

(Listings begin on page 110.)

Vote for your favorite feature/article.
Circle Reader Service **No. 8.**



The Power in PowerBasic

This new compiler may seem like an old friend

Bruce Tonkin

Earlier this year, Borland granted to Bob Zale, creator of Turbo Basic, rights to sell future versions of his compiler, which is now being marketed by Spectra Publishing as PowerBasic (PB). PowerBasic's designer has paid primary attention to the needs of the programmer while providing a Basic that is upwardly compatible with Turbo Basic 1.0 and Microsoft's GWBasic.

There are compromises in some areas, but there are very few when it comes to making programming easier and more productive. The result is a worthwhile and unusual version of Basic — but by no means non-standard.

Compromise of a Sort

Regardless of language, most programmers hate reinventing and rewriting the known. With languages such as C and Pascal, quite a lot of money goes into programmer's toolkits. Still, many common routines fall through the cracks: They're too short and too easy to be worth putting in a toolkit. Worse, some parts of the language itself make the programmer do more work — with no offsetting gain in clarity.

Consider a simple *sort* routine. If you were sorting many items, you'd dig out a routine from your favorite toolkit. Alas, the routine might be too general — requiring that you specify a sort order, a comparison function, data types, and so on. Modifying to sort a half dozen strings might be more trouble than it's worth. So you rewrite a bubble sort for the thousandth time. Sure, it's only a few lines of code, but it can be irritating (especially if you make a mistake in something that elementary).

Or suppose you need a large text array of variable-length strings. I've read quite a few articles about managing such things in C and Pascal, and I've reinvented many of the algorithms in Knuth's books for managing garbage collection. Lately, dynamic string libraries have begun to appear. I suppose they're useful, but I think the need for those libraries begs the question, "Why isn't the requisite functionality built into the language or the standard libraries already?"

QuickBasic and Microsoft's excellent Basic 7.0 still don't permit dynamic strings of more than 64K at one time. Basic 7.0 dodges the question somewhat by allowing the user to have several arrays of as much as 64K at one time. Still, no single array may have more than 64K of dynamic strings.

Fixed-length strings are standard for C and Pascal. There's no question that fixed-length strings are quite useful and even preferable to dynamic strings in many applications. It was an improvement when such strings were added to Basic. However, it's less useful if the length must be specified at compile-time rather than at run time. There are far too many instances (file utilities, sort programs, and so on) where the string length simply cannot be known in advance.

PB doesn't pretend to be an entirely new language, nor does it claim to solve all problems. It's not so much a paradigm shift as an exercise in pragmatism. To the question, "What should Basic be?" Bob Zale has answered: "Whatever Basic programmers want." He didn't implement everything I would like to see, but then, that may be impossible.

Yes, PB includes a *sort*. The command will sort all or parts of arrays of all data types (the default is plain ASCII), using any collating order you specify, either directly or using a tag array, in ascending or descending order. The sorted arrays may have many dimensions. The options are not mandatory parts of the command, so you can sort a whole array in ascending ASCII order simply as: ARRAY SORT A$( )

That's useful enough, but there's more. You can also scan all or part of an array for the first element that matches a relation you specify (<, >, = , etc.), based (for string arrays) on any collating order you specify — including case-insensitive scans.

You can also insert or delete an array element with a single command. No more loops or swaps are needed. It seems an enormous understatement to say that all this is just "useful" — it's been needed for years!

Bruce develops and sells software for TRS-80 and MS-DOS/PC-DOS computers. He can be reached at T.N.T. Software Inc., 34069 Hainesville Road, Round Lake, IL 60073.

(continued from page 72)

Language Enhancements

These and other enhancements are discussed in the two manuals (*User's Manual and Reference Guide*) supplied with PB. The documentation is thorough (more than 700 pages), clear, and well indexed and organized, and seems to be free from any obvious typos. I found no errors in the manuals while doing the review, but I'd suggest that later versions of the manuals devote an appendix to the differences between PB and QB. There are appendices detailing the differences between PB and GWBasic, and between Turbo Basic and PB.

Among other things, PB has new data types: Floating-point BCD, fixed-point BCD, 8-byte integer, and 10-byte extended precision floating-point (native coprocessor format numbers). 8-byte integers can have as many as 18 digits, so PRINT USING was revamped.

The expanded string space carries several penalties. Because strings can be spread over more memory, there's more overhead involved. That means string operations tend to be slower in PB than in Microsoft products. But if you need the space, the small penalty is well worth it.

In addition, the FRE(" ") function returns the amount of space remaining in the current string allocation block. Once you've been running the program for a while, there's no easy way for you to tell how much total memory is left for strings, or how much you've used so far. You must calculate the amount of available memory when the program first starts and use that number as the baseline for future calculations.

Those are problems to be noted, but they aren't serious, especially for those of us who find the unlimited string space to be a big advantage.

As with Turbo Basic, PB permits the programmer to specify where segment boundaries lie. From that, you can avoid the problems created by programs too large to fit into a single-user code segment. However, the PB editor and compiler limit source code to 64K bytes. This is less of a problem than before because PB can link with OBJ files and units (resembling Turbo Pascal's units). It can, however, be somewhat of a bother.

Options

Compilation options allow you to generate code for 80286 and 80386 processors; use/emulate a math coprocessor (or ignore the coprocessor and go for maximum speed with a procedural math library); turn on or off various checks; and reduce code size by eliminating support for unnecessary library modules. What you get seems very similar to the options provided in Microsoft's Basic 7.0, though PB's EXE files are usually larger than those of Basic 7.0.

PB permits linking OBJ modules in what seems to me a more natural way than with the Microsoft compilers: PB asks that you include the names of the OBJ modules within the source code of your program. That decreases the length of the command line and the possibility of errors. The compilation options can also be included as metacommands in the source code. Those enhancements make it much easier to compile PB programs.

There's no question that fixed-length strings are quite useful and even preferable to dynamic strings in many applications

There is a drawback, though. PB cannot link with libraries, as QuickBasic can. This can be a big problem if you need many OBJ files.

Benchmarks

While doing this review, I churned out my usual quantitative comparisons. BENCHOLD.BAS (Listing One, page 116) benchmarks features of the language that are common to QuickBasic 3.0 and earlier, while BENCHNEW.BAS (Listing Two, page 119) tests the newer features of the language. The results are shown in Tables 1 and 2. Those numbers are not completely irrelevant, but they are of less worth than usual: Power Basic breaks new ground in unexpected ways, and the overall utility of this compiler cannot be judged by noting the relative time it takes to perform a string concatenation or a double precision add.

True compilation is faster in PB. With the Microsoft products, any request to create an EXE will result in a call to the command line compiler and the separate linker. PB compiles in memory.

If you ask to create an EXE in memory, the Microsoft products are faster; they're not really compiling but converting the program to a tokenized form. The difference shows when you run the result. PB programs run at EXE speed, even in the environment. PB compiles slower because it's really compiling, but that doesn't mean it's a slug. Where QuickBasic might take 2 seconds to tokenize and start running a program, PB might take 10. The QB program might take 20 seconds to run in the environment, where the PB version might run in 5.

So, who's faster? It depends. If you often need to stop a program and rerun it with minor changes, QB might be best. But if the program is complex enough, PB might give a faster turnaround. The more complex the program, the bigger the advantage for PB. PB permits conditional compilation, too. This is a feature I'd hoped Microsoft would include in Basic 7.0.

Environmental Issues

PB's editor is much better than Microsoft's. The user interface (in my opinion) hurts the Microsoft products. The QB editor will not get out of the way, and constantly presents dialog boxes telling you of syntax errors while you're doing ordinary editing from line to line.

Another problem is that the QB editor has no line continuation character. Long lines must be viewed in parts or split into sections. This can be a nuisance because the QB editor insists on inserting spaces at nearly every opportunity. Even modestly complex lines can easily become too wide and can't be split. Additionally, the QB editor will not permit the programmer to mark

COMPILER	PROGRAM	SOURCE SIZE	EXE SIZE
POWER BASIC	BENCHNEW	3400	39296
QB 4.5	BENCHNEW	3022	34276 *
BASIC 7.0	BENCHNEW	3418	24848
POWER BASIC	BENCHOLD	8721	46384
QB 4.5	BENCHOLD	7746	37722 *
BASIC 7.0	BENCHOLD	8720	32684

**Supports fewer tests; size is not strictly comparable, but is provided for reference.*

Table 1: *Size of EXEs generated from Listings One and Two*

(continued from page 74)
a block of code and write it to a disk file, or to read a block from a disk file into the current program.

By default, the QB editor saves new programs in a special format. The format varies from release to release, and none are backwardly or upwardly compatible.

PB lacks support for OS/2 and doesn't have the ISAM file capability of Basic 7.0

In contrast, PB works with and saves plain ASCII files, alters nothing you type, permits line continuation characters, and allows block read and write. It's just plain easier and faster to type and edit code with PB.

To be fair, I find the QB help system easier to use than PB's. That may be a factor for beginners. And, QB's instant debugging is probably better for programmers not familiar with Basic. Again, there are definite trade-offs. PB's editor is perhaps better suited to programmers with some experience in Basic.

PB's debugging options are improved over Turbo Basic's, but Microsoft still has a clearly better approach. PB's debugger is competent but clumsier and somewhat more difficult to use.

PB's debugger doesn't permit you to use function calls, but only to change the value of a variable or an expression that evaluates to a variable. Further, PB replaces PRINT USING during debugging with a series of powerful but arcane formatting commands. The result is flexible and more capable than what Microsoft delivers, but I found it confusing. The commands are reminiscent of Fortran with some C conventions. Fortunately, for most debugging sessions you'll never need to worry about formatted print for variables.

Trade-offs

Another point that PB has in its favor is the physical size of the compiler environment. PB can be run easily on a floppy disk system. Many beginners don't have hard disks. Even if you installed every PB file on your hard disk, the total would come to just about 700K — much less than QuickBasic 4.5, and far less than the 6 to 14 Mbytes required for Basic 7.0.

Part of the reason for the size difference: PB lacks support for OS/2 and doesn't have the ISAM file capability of Basic 7.0. Nor does PB come with the equivalent of CodeView. But I have yet to find a use for CodeView when doing Basic programming.

PB doesn't support record or file locking, either. Of all the possible disadvantages I can imagine, this is probably the most serious. While OS/2 is very

Time, in seconds, per 1,000,000 operations

	QB 3.0	QB 4.0	BASIC 6	BASIC 7	PB
Integers:					
Empty loop	1.21	1.21	1.21	1.20	2.09
assignment:	1.15	1.13	1.13	1.10	1.02
add:	.30	.49	.50	.27	.47
subtract:	.33	.46	.49	.16	.77
multiply:	1.92	2.45	2.27	2.25	2.77
divide:	3.36	3.43	3.44	3.90	5.41
comparison:	2.58	3.20	2.58	2.53	2.91
Conditional assignment:	4.51	4.77	4.67	2.37	2.31
Conditional assignment*:	4.72	4.88	4.34	1.70	2.47
Long integers:					
Empty loop	-----	14.45	14.28	12.59	10.49
assignment:	-----	1.62	1.71	1.53	1.67
add:	-----	15.47	15.26	13.23	11.73
subtract:	-----	15.41	15.21	13.57	12.22
multiply:	-----	26.11	25.82	24.34	25.35
divide:	-----	33.76	33.66	31.70	25.68
comparison:	-----	10.81	10.98	10.48	8.57
Single-precision:					
assignment:	5.71	201.23	20.40	2.20	1.98
add:	23.97	128.83	24.41	42.45	24.66
subtract:	24.80	129.36	25.87	43.72	26.34
multiply:	34.30	-16.19	35.50	52.54	36.02
divide:	35.98	7.48	47.68	64.80	47.02
Error,100K mult/div:	-1.06E-05	-1.19E-07	-1.19E-07	-1.19E-07	-1.96E-5
exponential:	766.80	3061.84	1296.36	1309.90	3373.80
comparison:	19.72	402.00	42.68	42.58	4.55
Double-precision:					
assignment:	6.26	211.41	23.06	3.49	49.10
add:	45.75	143.28	49.99	68.80	69.48
subtract:	47.19	143.87	51.52	70.39	69.43
multiply:	83.13	199.49	91.66	109.60	86.13
divide:	84.22	226.45	121.82	142.57	130.08
Error, 100K mult/div:	-4.82E-13	-2.22E-16	2.22E-16	2.22E-16	-4.22E-15
exponential:	2491.80	6377.23	3046.33	3240.62	3326.56
comparison:	20.37	412.89	46.30	46.37	100.68
Strings:					
assignment:	77.09	78.50	79.04	77.72	72.56
MID$ operation:	20.78	24.61	25.20	24.11	81.93
concatenation:	1657.81	954.84	1661.15	1662.48	875.16
Print 1K 70-byte strings**:	26.47	10.00	10.05	9.55	9.61
Fixed string assignment:	-----	52.38	51.92	51.68	101.86
Fixed string MID$ operation:	-----	23.44	24.12	23.74	147.13
Fixed string concatenation:	-----	24.61	24.78	21.01	170.85
Pr 1K 70-b static strings**:	-----	10.16	10.10	9.65	9.60

Table 2: *Timings were generated using a Tandy 4000 (16-MHz 80386) with an MDA video adapter card, a Casper amber monochrome monitor, and no math coprocessor. For each compiler, all possible stub libraries were used, all error checks were removed, and all speed optimizations employed.*

much a minor market, LANs are everywhere on the increase. So far, though, fewer than 20 percent of MS-DOS machines are part of a network. Of that minority, many do not need shared file access through a user-written program; a large number share files only through networked word processors or the like. Realistically, PB's potential market should not be damaged greatly by the lack of support for shared data files. It is, however, something that I recommend be added soon.

(continued on page 80)

**One of the ways should be appreciatively faster if short-circuit optimization is being done*
***To the screen. Monochrome display, MDA card*
Note: the negative and small times for single-precision multiply and divide for QB 4.0 mean that they took less time to execute than a simple assignment. Perhaps there are unnecessary checking or conversion operations going on during the simple assignment. Considering the big slow-down from QB 3.0, that seems likely.

Operation speed QuickBasic 3.0 = 100.0 (Lower numbers are better)

	QB 4.0	BASIC 6	BASIC 7	PB
Empty integer loop	100.0	100.0	99.2	172.7
Integer assignment:	98.3	98.3	95.7	88.7
Integer add:	163.3	166.7	90.0	156.7
Integer subtract:	139.4	148.5	48.5	233.3
Integer multiply:	127.6	118.2	117.2	144.3
Integer divide:	102.1	102.4	116.1	161.0
String assignment:	101.8	102.5	100.8	94.1
String MID$ operation:	118.4	121.3	116.0	394.3
String concatenation:	57.6	100.2	100.3	113.1
Single-precision assignment:	3524.2	357.3	38.5	34.7
Single-precision add:	537.5	101.8	177.1	102.9
Single-precision subtract:	521.6	104.3	176.3	106.2
Single-precision multiply:	-47.2	103.5	153.2	105.0
Single-precision divide:	7.4	47.6	180.1	130.7
Single-precision exponential:	399.3	169.1	170.8	440.0
Double-precision assignment:	3377.2	368.4	55.8	784.3
Double-precision add:	313.2	109.3	150.4	151.9
Double-precision subtract:	304.9	109.2	149.2	147.1
Double-precision multiply:	240.0	110.3	131.8	103.6
Double-precision divide:	268.9	144.6	169.3	154.5
Double-precision exponential:	255.9	122.3	130.1	133.5
Integer comparison:	124.0	100.0	98.1	112.8
Single-precision comparison:	402.0	216.4	215.9	73.8
Double-precision comparison:	2027.0	227.3	227.6	494.3
Conditional int assignment:	105.8	103.5	52.5	51.2
Conditional assignment*:	103.4	91.9	36.0	52.3
Print 1K 70-byte strings**:	37.8	38.0	36.1	36.3
Unweighted average performance:	500.4	136.4	119.7	176.8

Operation speed, Basic 6.0 = 100.0

	QB 4.0	BASIC 7	PB
Empty long integer loop	101.2	88.2	73.5
Long integer assignment:	94.7	89.5	97.7
Long integer add:	101.4	86.7	76.9
Long integer subtract:	101.3	89.2	80.3
Long integer multiply:	101.1	94.3	98.2
Long integer divide:	100.3	94.2	76.3
Fixed string assignment:	100.9	99.5	196.2
Fixed string MID$ operation:	97.2	98.4	610.0
Fixed string concatenation:	99.3	84.8	689.5
Long integer comparison:	98.5	95.4	78.1
Print 1K 70-byte static strings**:	100.6	95.5	95.0
Unweighted performance average:	99.7	92.3	197.4

(continued from page 77)

Putting It to Work

To demonstrate some of PowerBasic's new features, consider the simple mailing list program in Listing Three (page 120). The program certainly doesn't exercise all the capabilities of PB, but it should be enough to give you a taste of them.

Listing Three doesn't contain a search function, holds all data in memory, and has primitive editing capabilities (you can delete a record and insert a new one, and that's it). You can, however, hold more than 3100 records in memory on a 640K machine (473,176 bytes available for string data). The *sort* is fast, record insertion is nearly instantaneous, and it would be trivial to add features for finding a record, editing by field, sorting by additional fields, retaining a backup data file, and so on. In fact, most things you might want to do to this program are easy.

Conclusion

PB's new features are a strong selling point. Its emphasis on genuinely useful enhancements, the nice editor, the faster execution in the environment, and the new data types make it an essential product for anyone seriously interested in Basic.

Neither PB nor Microsoft's Basic 7.0 is the "best" Basic compiler in every area. PB needs support for libraries, file and record locking, larger source files, user-defined types, and a better debugger. Microsoft could benefit from PB's editor and environment, new functions, larger string space, link and metacommand options, conditional compilation, and added data types.

I intend to convert several of my programs from QB and Basic 7.0 to PB. In fact, I began the process soon after getting my review copy of PB. PB's advantages in particular areas make the conversion attractive. Other programs will remain in the Microsoft dialect. I suspect that most Basic programmers will have the same experience.

If you're serious about Basic programming, you really ought to consider having both PB and one of the Microsoft products. And Turbo Basic 1.x owners will find the $59 upgrade is money well spent.

Product Information

PowerBasic
Spectra Publishing
1030-D E. Duane
Sunnyvale, CA 94086
408-730-9291
Price: $129
Upgrade for Turbo Basic users $59
Requirements: IBM PC/compatible
MS-DOS 2.0 or higher
640K RAM, one floppy drive

Source Code Availability

As a service to our readers, all source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Drive, Redwood City, CA 94063, or call 800-356-2002 (inside Calif.) or 800-533-4372 (outside Calif.). Specify issue number and disk format. Code is also available through the DDJ Forum on CompuServe (type GO DDJ).

DDJ

(Listings begin on page 116.)

Vote for your favorite feature/article.
Circle Reader Service **No. 9.**

Listing One *(Text begins on page 16.)*

```
; svDefs.INC                                                          ;
; Copyright (c) Genus Microprogramming, Inc. 1988-89  All Rights Reserved.  ;
;*************************************************************************;
; This file contains defines for Super VGA graphics manipulation.       ;
; Microsoft ASM 5.x version.              Programmer: Chris Howard      ;
;*************************************************************************;

;Display Segments
EGAseg    equ       0A000H                ;EGA/VGA/SVGA graphics segment
BIOSseg   equ       0C000H                ;Graphics BIOS segment

;EGA defines
EGAgraph  equ       03CEH                 ;EGA Graphics Register
EGAseq    equ       03C4H                 ;EGA Sequencer Register

;VGA defines
VGAsegsel equ       03CDH                 ;VGA Segment select
VGAmisci  equ       03CCH                 ;VGA Misc In
VGAmisco  equ       03C2H                 ;VGA Misc Out

;Paradise defines
parPROA   equ       09H                   ;proa index value
parPR5    equ       0FH                   ;pr5  index value
parLOCK   equ       0                     ;Lock   proa to pr4 (write to pr5)
parUNLOCK equ       5                     ;Unlock proa to pr4 (write to pr5)
parFUNC   equ       6FH                   ;Paradise Function

;Video 7 defines
v7SR6     equ       06H                   ;sr6 index value
v7banksel equ       0F6H                  ;Bank select
v7pagesel equ       0F9H                  ;Page select
v7enable  equ       0EAH                  ;Enable extensions
v7disable equ       0AEH                  ;Disable extensions
v7modenum equ       06F05H                ;Number to use for mode sets

;Display mode numbers
TEXTMODE  equ       3                     ;Text mode number

;Display mode types
svTEXT    equ       0                     ;Text mode
svGRAPH   equ       1                     ;Graphics mode

;Display types
mindisp   equ       0                     ;Minimum display type
svDISP_2D equ       0                     ;Tseng    2DH  (640x350x256)
svDISP_2E equ       1                     ;Tseng    2EH  (640x480x256)
svDISP_30 equ       2                     ;Tseng    30H  (800x600x256)
svDISP_5E equ       3                     ;Paradise 5EH  (640x400x256)
svDISP_5F equ       4                     ;Paradise 5FH  (640x480x256)
svDISP_66 equ       5                     ;Video 7  66H  (640x400x256)
svDISP_67 equ       6                     ;Video 7  67H  (640x480x256)
svDISP_69 equ       7                     ;Video 7  69H  (800x600x256)
maxdisp   equ       7                     ;Maximum display type

;Logical Operations
RMWbits   equ       18H                   ;Read-Modify-Write bits
svOpREP   equ       00000000B             ;SET pixel value directly
svOpAND   equ       00000001B             ;AND pixel value with data
svOpOR    equ       00000010B             ;OR  pixel value with data
svOpXOR   equ       00000011B             ;XOR pixel value with data

;Chipsets
svUNKNOWN equ       0                     ;Unknown chip set
svTSENG   equ       1                     ;Tseng Labs chip set
svPARA    equ       2                     ;Paradise
svV7      equ       3                     ;Video 7

;Masks
capmask   equ       11011111B             ;To convert letters to caps

;Error codes
svSUCCESS equ       0                     ;Success
svBADMODE equ       -1                    ;Bad display mode

;Internal Constants
unknown   equ       -1                    ;A constant is unknown
bytesrow  equ       80                    ;CGA bytes per full row
bitsbyte  equ       8                     ;Bits per byte

;Display structure
svstruc   STRUC
svtype    db        ?                     ;Display type
svmode    db        ?                     ;Display mode
svfunc    dd        ?                     ;Display function
svstruc   ENDS
```

End Listing One

Listing Two

```
; svMacs.INC                                                          ;
; Copyright (c) Genus Microprogramming, Inc. 1988-89  All Rights Reserved.  ;
;*************************************************************************;
; This file contains macros for Super VGA graphics manipulation.        ;
; Microsoft ASM 5.x version.              Programmer: Chris Howard      ;
;*************************************************************************;
; Interrupts
INT_BIOS  equ       10H                   ;BIOS (video)
INT_DOS   equ       21H                   ;DOS Functions

; BIOS Functions (int 10H)
SETMODE   equ       00H                   ;Set the Display mode
GETMODE   equ       0FH                   ;Check the Display mode

; Macros
@DOS      MACRO     func                  ;;Macro to use DOS function calls
          mov       ah,func
          int       INT_DOS
          ENDM

@BIOS     MACRO     func                  ;;Macro to use BIOS function calls
          push      bp                    ;;Some functions destroy bp
          mov       ah,func
          int       INT_BIOS
          pop       bp
          ENDM

@@LoadSeg MACRO     seg,val               ;;Macro to load a segment reg
          mov       ax,val
          mov       seg,ax
          ENDM

@@Data    MACRO     seg                   ;;Macro to load data seg in reg
          ASSUME    seg:@data
          mov       ax,@data
          mov       seg,ax
          ENDM

@SetMode  MACRO     mode                  ;;Macro to set display mode
          IFDIFI    <mode>,<al>
          mov       al,mode
          ENDIF
          @BIOS     SETMODE
          ENDM

@GetMode  MACRO                           ;;Macro to get display mode
          @BIOS     GETMODE
          ENDM

@Port     MACRO     portnum,portval       ;;Macro to set a port to a value
          IFDIFI    <portval>,<al>
          mov       al,portval
          ENDIF
          IFDIFI    <portnum>,<dx>
          mov       dx,portnum
          ENDIF
          out       dx,al
          ENDM

@EGAPort  MACRO     portnum,portreg,portval ;;Macro to set an EGA port
          IFDIFI    <portreg>,<al>
          mov       al,portreg
          ENDIF
          IFDIFI    <portnum>,<dx>
          mov       dx,portnum
          ENDIF
          out       dx,al
          inc       dx
          mov       al,portval
          out       dx,al
          ENDM

@TsengSeg MACRO     seg                   ;;Macro to set the Tseng VGA seg
          push      ax
          IFDIFI    <seg>,<al>
          mov       al,seg
          ENDIF
          mov       ah,al                 ;;Save a copy
          shl       ah,1                  ;;Rotate up (Bits 0-2 = write seg
          shl       ah,1                  ;;                3-5 = read  seg
          shl       ah,1                  ;;                6-7 = seg cnfg)
          or        al,ah                 ;;Combine read and write segs
          or        al,01000000B          ;; and set to configuration 2
          @Port     VGAsegsel,al          ;;Now set it
          pop       ax
          ENDM

@ParSeg   MACRO     seg                   ;;Macro to set Paradise VGA seg
          push      cx
          IFDIFI    <seg>,<ch>
          mov       ch,seg
          ENDIF
          mov       cl,4                  ;Turn 4K window into 64K window
          shl       ch,cl                 ; by multiplying by 16
          @EGAPort  EGAgraph,parPROA,ch   ; and set the new index
          pop       cx
          ENDM

@V7Seg    MACRO     seg                   ;;Macro to set the V7VGA segment
          push      ax
          push      bx
          IFDIFI    <seg>,<bh>
          mov       bh,seg
          ENDIF
          mov       bl,bh
          and       bl,00000001B          ;;Mask for bank bit 0
          @EGAPort  EGAseq,V7pagesel,bl   ;;Set the bit
          mov       bl,bh                 ;;Get a copy of seg again
          and       bl,000000010B         ;;Mask for bank bit 1
          shl       bl,1                  ;;Shift it up
          shl       bl,1
          shl       bl,1
          shl       bl,1
          mov       dx,VGAmisci           ;;Read the Misc In reg
          in        al,dx
          and       al,NOT 00100000B      ;;Make sure bit is clear
          or        bl,al
          @Port     VGAmisco,bl           ;;Write to Misc Out
          @Port     EGAseq,v7banksel
          inc       dx
          in        al,dx
          mov       bl,bh                 ;;Get a copy of seg again
          shr       bl,1                  ;;Dupe bit 2 to bit 0 (r/w equal)
          shr       bl,1
          add       bl,7
          not       bl
          and       bl,5
          and       al,11110000B          ;;Clear bank select bits
          or        al,bl
          out       dx,al
```

(Listing continued on page 84.)

Listing Two *(Listing continued, text begins on page 16.)*

```
        pop     bx
        pop     ax
        ENDM

@V7Mode MACRO   mode                    ;;Macro to set Video7 mode
        IFDIFI  <mode>,<bl>
        mov     bl,mode
        ENDIF
        mov     ax,v7modenum            ;;Indicate Video7 mode set
        int     INT_BIOS
        ENDM

@Entry  MACRO   splocal                 ;;Macro for entering routine
        push    bp                      ;; and setting up frame
        mov     bp,sp
        sub     sp,splocal              ;;Allocate local space
        push    ds
        push    es
        push    si
        push    di
        @@Data  ds
        ENDM

@Exit   MACRO   retcode,splocal         ;;Macro for setting return code
        mov     ax,retcode              ;; and restoring regs
        pop     di
        pop     si
        pop     es
        pop     ds
        mov     sp,bp                   ;;Remove local space
        pop     bp                      ;;Restore frame
        ret     splocal                 ;;Remove parms
        ENDM

@SetRet MACRO   retcode,errcode
        mov     WORD PTR retcode,errcode
        ENDM
```

End Listing Two

Listing Three

```
; svQC.ASM                                                                ;
; Copyright (c) Genus Microprogramming, Inc. 1988-89  All Rights Reserved.  ;
;*************************************************************************;
; This file contains procedures for identifying various Super VGA adapters.  ;
; Procedures: svQueryChipset                                              ;
; Microsoft ASM 5.x version.              Programmer: Chris Howard        ;
;*************************************************************************;

; Include files
  INCLUDE svDefs.inc
  INCLUDE svMacs.inc

  .model  small
  .data
  .code
        PUBLIC    svQueryChipset

;**********
; This procedure attempts to determine the type of VGA chip set.
; Calling: retcode = pcxQueryChipset()
;

;Define variable locations on the stack  (pascal model)
qcparm  equ       0

;Define local variables
qcret   equ       <[bp- 2]>             ;return code
qclocal equ       2                     ;Total local space needed

svQueryChipset    PROC FAR
        @Entry    qclocal               ;Set up frame and save regs
        mov       al,0                  ;Look for a V7VGA
        @BIOS     parFUNC
        xor       al,al                 ;Clear al
        cmp       bx,'V7'               ;Is this a Video7?
        jne       svQC_Para
        @SetRet   qcret,svV7            ;Indicate that this is a Video7
        jmp       svQC_exit

svQC_Para:
        @@LoadSeg es,BIOSseg            ;Point to the BIOS location
        mov       di,0
        mov       cx,500                ;Search the first 500 bytes
        mov       al,'P'                ;Look for 'PARADISE'

svQC_Parafind:
        repne     scasb                 ;Search
        jcxz      svQC_Tseng
        cmp       BYTE PTR es:[di  ],'A'  ;Next Letter?
        jne       svQC_Parafind         ; Compare one at a time, so
                                        ; we can avoid static data ...)
        cmp       BYTE PTR es:[di+1],'R'  ;
        jne       svQC_Parafind         ; If not a match, continue search
        cmp       BYTE PTR es:[di+2],'A'
        jne       svQC_Parafind
        cmp       BYTE PTR es:[di+3],'D'
        jne       svQC_Parafind
        cmp       BYTE PTR es:[di+4],'I'
        jne       svQC_Parafind
        cmp       BYTE PTR es:[di+5],'S'
        jne       svQC_Parafind
        cmp       BYTE PTR es:[di+6],'E'
        jne       svQC_Parafind

        @SetRet   qcret,svPARA          ;We found the Paradise name
        jmp       SHORT svQC_exit
```

```
svQC_Tseng:
         @@LoadSeg es,BIOSseg             ;Point to the BIOS location
         mov       di,0
         mov       cx,500                 ;Search the first 500 bytes
         mov       al,'T'                 ;Look for 'Tseng'

svQC_Tsfind:
         repne     scasb                  ;Search
         jcxz      svQC_unknown

         cmp       BYTE PTR es:[di  ],'s' ;Next Letter?
         jne       svQC_Tsfind            ; Compare one at a time, so
                                          ; we can avoid static data ...)
         cmp       BYTE PTR es:[di+1],'e' ;
         jne       svQC_Tsfind            ; If not a match, continue search
         cmp       BYTE PTR es:[di+2],'n'
         jne       svQC_Tsfind
         cmp       BYTE PTR es:[di+3],'g'
         jne       svQC_Tsfind

         @SetRet   qcret,svTSENG          ;We found the Tseng name
         jmp       SHORT svQC_exit

svQC_unknown:
         @SetRet   qcret,svUNKNOWN        ;We did not find anything

svQC_exit:
         @Exit     qcret,qcparm           ;Return
svQueryChipset    ENDP
         END
```

End Listing Three

Listing Four

```
; svPA.ASM                                                                    ;
; Copyright (c) Genus Microprogramming, Inc. 1988-89  All Rights Reserved.    ;
;*****************************************************************************;
; This file contains procedures for calculating a pixel's address for any     ;
; given display mode. These are INTERNAL routines.                            ;
; Procedures: svPixelAddr2D  svPixelAddr30  svPixelAddr5F                     ;
; Microsoft ASM 5.x version.              Programmer: Chris Howard            ;
;*****************************************************************************;

; Include files
  INCLUDE svDefs.inc
  INCLUDE svMacs.inc

  .model  small
  .data
  .code
         PUBLIC    svPixelAddr2D
         PUBLIC    svPixelAddr30

;**********

; This function determines the address of a pixel in SVGA 256 color
; modes:
;          2DH     640x350x256 Tseng
;          2EH     640x480x256 Tseng
;          66H     640x400x256 Video7
;          67H     640x480x256 Video7
;          5EH     640x400x256 Paradise
;          5FH     640x480x256 Paradise
; Calling: AX    = y-coordinate
;          BX    = x-coordinate
; Returns: ES:BX = pixel pointer
;          DX    = video segment
;

svPixelAddr2D       PROC FAR

         mov       dx,640                 ;Multiply y*BytesPerLine
         mul       dx
         add       bx,ax                  ;Add in x coordinate
         adc       dx,0                   ; and any carry
         @@LoadSeg es,EGAseg              ;ES:BX = byte address of pixel
         ret

svPixelAddr2D       ENDP

;**********

; This function determines address of pixel in SVGA 800x600x256 color modes:
;          30H     800x600x256 Tseng
;          69H     800x600x256 Video7
; Calling: AX    = y-coordinate (0-599)
;          BX    = x-coordinate (0-799)
; Returns: ES:BX = pixel pointer
;          DX    = video segment
;

svPixelAddr30       PROC FAR

         mov       dx,800                 ;Multiply y*bytesrow
         mul       dx
         add       bx,ax                  ;Add in x coordinate
         adc       dx,0                   ; and any carry
         @@LoadSeg es,EGAseg              ;ES:BX = byte address of pixel
         ret

svPixelAddr30       ENDP

         END
```

End Listing Four

(continued on page 86)

Listing Five *(Listings continued, text begins on page 16.)*

```
; svPP.ASM                                                              ;
; Copyright (c) Genus Microprogramming, Inc. 1988-89  All Rights Reserved.  ;
;************************************************************************;
; This file contains procedures for putting (displaying) a pixel for any    ;
; given display mode.                                                    ;
; Procedures: svPutPixel                                                 ;
; Microsoft ASM 5.x version.              Programmer: Chris Howard       ;
;************************************************************************;

; Include files
  INCLUDE svDefs.inc
  INCLUDE svMacs.inc

  .model  small
  .data

svDisplay svstruc   <svDISP_2D,2DH,svPutPixel2D>      ;Tseng Labs
svlen     EQU       $-svDisplay
          svstruc   <svDISP_2E,2EH,svPutPixel2D>
          svstruc   <svDISP_30,30H,svPutPixel30>
          svstruc   <svDISP_5E,5EH,svPutPixel5F>      ;Paradise
          svstruc   <svDISP_5F,5FH,svPutPixel5F>

          svstruc   <svDISP_66,1AH,svPutPixel67>      ;Video7
          svstruc   <svDISP_67,1BH,svPutPixel67>
          svstruc   <svDISP_69,1DH,svPutPixel69>

          PUBLIC    svCurDisp,svLogOp

svCurDisp dw        svDISP_2D               ;Current display type
svPixFunc dd        ?                       ;Current pixel function
svBank    db        -1                      ;Current window bank
svLogOp   db        svOpREP                 ;Logical operation
          EXTRN     svPixelAddr2D           : FAR
          EXTRN     svPixelAddr30           : FAR
  .code
          PUBLIC    svSetDisplay
          PUBLIC    svSetMode
          PUBLIC    svSetOp
          PUBLIC    svPutPixel
;**********

; This function sets the display type by selecting the correct pixel
; function for all svPutPixel calls.

;Define variable locations on the stack  (pascal model)
sddisp    equ       <[bp+ 6]>               ;Display type
sdparm    equ       2

;Define local variables
sdret     equ       <[bp- 2]>               ;return code
sdlocal   equ       2                       ;Total local space needed

svSetDisplay        PROC FAR

          @Entry    sdlocal                 ;Set up frame and save regs
          @@Data    es                      ;Point to table
          mov       ax,sddisp               ;Get the display type
          mov       svCurDisp,ax            ;Assume valid, and store
          mov       bx,svlen                ;Get offset into table
          mul       bx
          mov       di,OFFSET svDisplay     ;Add starting location of table
          add       di,ax
          mov       ax,WORD PTR es:[di].svfunc[2]  ;Store current function
          mov       WORD PTR svPixFunc[2],ax       ; for fast reference
          mov       ax,WORD PTR es:[di].svfunc[0]
          mov       WORD PTR svPixFunc[0],ax

          @SetRet   sdret,svSUCCESS

svSD_exit:
          @Exit     sdret,sdparm

svSetDisplay        ENDP

;**********

; This function sets graphics mode for the currently selected display type.

;Define variable locations on the stack  (pascal model)
smmode    equ       <[bp+ 6]>               ;Flag for TEXT or GRAPHICS mode
smparm    equ       2

;Define local variables
smret     equ       <[bp- 2]>               ;return code
smlocal   equ       2                       ;Total local space needed

svSetMode           PROC FAR

          @Entry    smlocal                 ;Set up frame and save regs
          mov       svBank,-1               ;Initialize bank
          mov       ax,smmode               ;Get requested "mode"
          cmp       ax,svGRAPH              ;Setting to graphics?
          je        svSM_graph
          @SetMode  TEXTMODE                ;Set to text mode
          @SetRet   smret,svSUCCESS
          jmp       SHORT svSM_exit

svSM_graph:
          @@Data    es                      ;Point to table
          mov       ax,svCurDisp            ;Get the display type
          mov       bx,svlen                ;Get offset into table
          mul       bx
          mov       di,OFFSET svDisplay     ;Add starting location of table
          add       di,ax
          @SetMode  es:[di].svmode          ;Set to correct graphics mode
          @GetMode                          ;Make sure it stuck
```

```
        cmp     al,es:[di].svmode
        je      svSM_ok
        @SetRet smret,svBADMODE         ;No, so return error
        jmp     SHORT svSM_exit

svSM_ok:
        @SetRet smret,svSUCCESS
svSM_exit:
        @Exit   smret,smparm

svSetMode       ENDP

;**********

; This function sets the logical operation for all svPutPixel calls.
;Define variable locations on the stack  (pascal model)
soop    equ     <[bp+ 6]>               ;Logical Operation
soparm  equ     2

;Define local variables
soret   equ     <[bp- 2]>               ;return code
solocal equ     2                       ;Total local space needed
svSetOp         PROC FAR

        @Entry  solocal                 ;Set up frame and save regs
        mov     ax,soop                 ;Get the logical operation
        cmp     ax,svOpXOR              ;Check range
        jbe     svSO_store
        @SetRet soret,svBADMODE         ;Error
        jmp     SHORT svSO_exit

svSO_store:
        mov     svLogOp,al              ;Store it
        @SetRet soret,svSUCCESS

svSO_exit:
        @Exit   soret,soparm

svSetOp         ENDP

;**********

; This function is the main entry point to all of the specific PutPixel
; routines. It sets up the appropriate parameters, then branches to the
; correct routine for the current display device.

;Define variable locations on the stack  (pascal model)
ppx     equ     <[bp+10]>               ;Pixel coordinate
ppy     equ     <[bp+ 8]>
ppcolor equ     <[bp+ 6]>               ;Color
ppparm  equ     6

;Define local variables
ppret   equ     <[bp- 2]>               ;return code
pplocal equ     2                       ;Total local space needed

svPutPixel      PROC FAR

        @Entry  pplocal                 ;Set up frame and save regs
        mov     ax,ppx                  ;Call pixel function
        push    ax
        mov     ax,ppy
        push    ax
        mov     ax,ppcolor
        push    ax
        call    DWORD PTR svPixFunc
        @SetRet ppret,ax

svPP_exit:
        @Exit   ppret,ppparm

svPutPixel      ENDP

;**********

; NOTE: The stack frame is defined in svPutPixel

svPutPixel2D    PROC FAR

        @Entry  pplocal
        mov     ax,ppy                  ;Set up call to address routine
        mov     bx,ppx
        call    svPixelAddr2D           ;ES:BX -> buffer, DL -> seg
        cmp     dl,svBank               ;Is bank currently selected?
        je      svPP2D_op

        @TsengSeg dl

svPP2D_op:

        mov     al,ppcolor              ;Get color
        mov     dl,svLogOp              ;Get operation
        cmp     dl,svOpRep
        jz      svPP2D_rep              ; (fastest if replace)
        cmp     dl,svOpXOR              ;Is this XOR?
        je      svPP2D_xor
        cmp     dl,svOpAND              ;Is this AND?
        je      svPP2D_and

svPP2D_or:
        or      es:[bx],al              ;Or the pixel
        jmp     short svPP2D_ok

svPP2D_and:
        and     es:[bx],al              ;And the pixel
        jmp     short svPP2D_ok

svPP2D_xor:
        xor     es:[bx],al              ;Routine to XOR
        jmp     short svPP2D_ok
```

(Listing continued on page 88.)

Listing Five *(Listing continued, text begins on page 16.)*

```
svPP2D_rep:
        mov     es:[bx],al              ;Set the pixel value

svPP2D_ok:
        @TsengSeg 0                     ;Reset
        @SetRet ppret,svSUCCESS
        @Exit   ppret,ppparm

svPutPixel2D    ENDP

;**********

; NOTE: The stack frame is defined in svPutPixel

svPutPixel30    PROC FAR

        @Entry  pplocal
        mov     ax,ppy                  ;Set up call to address routine
        mov     bx,ppx
        call    svPixelAddr30           ;ES:BX -> buffer, DL -> seg
        cmp     dl,svBank               ;Is bank currently selected?
        je      svPP30_op
        @TsengSeg dl

svPP30_op:
        mov     al,ppcolor              ;Get color
        mov     dl,svLogOp              ;Get operation
        cmp     dl,svOpRep
        jz      svPP30_rep              ; (fastest if replace)
        cmp     dl,svOpXOR              ;Is this XOR?
        je      svPP30_xor
        cmp     dl,svOpAND              ;Is this AND?
        je      svPP30_and

svPP30_or:
        or      es:[bx],al              ;Or the pixel
        jmp     short svPP30_ok

svPP30_and:
        and     es:[bx],al              ;And the pixel
        jmp     short svPP30_ok

svPP30_xor:
        xor     es:[bx],al              ;Routine to XOR
        jmp     short svPP30_ok

svPP30_rep:
        mov     es:[bx],al              ;Set the pixel value

svPP30_ok:
        @TsengSeg 0                     ;Reset
        @SetRet ppret,svSUCCESS
        @Exit   ppret,ppparm

svPutPixel30    ENDP

;**********

; NOTE: The stack frame is defined in svPutPixel

svPutPixel15F   PROC FAR

        @Entry  pplocal
        @EGAPort EGAgraph,parPR5,parUNLOCK  ;Unlock proa to pr4

        mov     ax,ppy                  ;Set up call to address routine
        mov     bx,ppx
        cmp     dl,svBank               ;Is bank currently selected?
        je      svPP5F_op
        call    svPixelAddr2D           ;ES:BX -> buffer, DL -> seg
        @ParSeg dl

svPP5F_op:
        mov     al,ppcolor              ;Get color
        mov     dl,svLogOp              ;Get operation
        cmp     dl,svOpRep
        jz      svPP5F_rep              ; (fastest if replace)
        cmp     dl,svOpXOR              ;Is this XOR?
        je      svPP5F_xor
        cmp     dl,svOpAND              ;Is this AND?
        je      svPP5F_and

svPP5F_or:
        or      es:[bx],al              ;Or the pixel
        jmp     short svPP5F_ok

svPP5F_and:
        and     es:[bx],al              ;And the pixel
        jmp     short svPP5F_ok

svPP5F_xor:
        xor     es:[bx],al              ;Routine to XOR
        jmp     short svPP5F_ok

svPP5F_rep:
        mov     es:[bx],al              ;Set the pixel value

svPP5F_ok:
        @EGAPort EGAgraph,parPROA,0     ;Zero out proa
        @EGAPort EGAgraph,parPR5,parLOCK ;Lock proa to pr4
        @SetRet ppret,svSUCCESS
        @Exit   ppret,ppparm

svPutPixel15F   ENDP

;**********
; NOTE: The stack frame is defined in svPutPixel

svPutPixel67    PROC FAR

        @Entry  pplocal
        @EGAPort EGAseq,v7SR6,v7enable  ;Enable Video 7 extensions

        mov     ax,ppy                  ;AX = y
        mov     bx,ppx                  ;BX = x
        call    svPixelAddr2D           ;ES:BX -> buffer, DL -> seg
        cmp     dl,svBank               ;Is bank currently selected?
        je      svPP67_op
        mov     cl,dl
        @V7Seg  cl

svPP67_op:
        mov     al,ppcolor              ;Get color
        mov     dl,svLogOp              ;Get operation
        cmp     dl,svOpRep
        jz      svPP67_rep              ; (fastest if replace)
        cmp     dl,svOpXOR              ;Is this XOR?
        je      svPP67_xor
        cmp     dl,svOpAND              ;Is this AND?
        je      svPP67_and

svPP67_or:
        or      es:[bx],al              ;Or the pixel
        jmp     short svPP67_ok

svPP67_and:
        and     es:[bx],al              ;And the pixel
        jmp     short svPP67_ok

svPP67_xor:
        xor     es:[bx],al              ;Routine to XOR
        jmp     short svPP67_ok

svPP67_rep:
        mov     es:[bx],al              ;Set the pixel value

svPP67_ok:
        @V7Seg  0                       ;Reset
        @EGAPort EGAseq,v7SR6,v7disable ;Disable Video 7 extensions
        @SetRet ppret,svSUCCESS
        @Exit   ppret,ppparm

svPutPixel67    ENDP

;**********

; NOTE: The stack frame is defined in svPutPixel

svPutPixel69    PROC FAR

        @Entry  pplocal
        @EGAPort EGAseq,v7SR6,v7enable  ;Enable Video 7 extensions
```

(Listing continued on page 90.)

Listing Five *(Listing continued, text begins on page 16.)*

```
        mov     ax,ppy                  ;Set up call to address routine
        mov     bx,ppx
        call    svPixelAddr30           ;ES:BX -> buffer, DL -> seg
        cmp     dl,svBank               ;Is bank currently selected?
        je      svPP69_op
        mov     cl,dl
        @V7Seg  cl

svPP69_op:
        mov     al,ppcolor              ;Get color
        mov     dl,svLogOp              ;Get operation
        cmp     dl,svOpRep
        jz      svPP69_rep              ; (fastest if replace)
        cmp     dl,svOpXOR              ;Is this XOR?
        je      svPP69_xor
        cmp     dl,svOpAND              ;Is this AND?
        je      svPP69_and

svPP69_or:
        or      es:[bx],al              ;Or the pixel
        jmp     short svPP69_ok

svPP69_and:
        and     es:[bx],al              ;And the pixel
        jmp     short svPP69_ok

svPP69_xor:
        xor     es:[bx],al              ;Routine to XOR
        jmp     short svPP69_ok

svPP69_rep:
        mov     es:[bx],al              ;Set the pixel value

svPP69_ok:
        @V7Seg  0                       ;Reset
        @EGAPort EGAseq,v7SR6,v7disable ;Disable Video 7 extensions
        @SetRet ppret,svSUCCESS
        @Exit   ppret,ppparm

svPutPixel69    ENDP

        END
```

End Listing Five

Listing Six

```
/* svLib.H                                                                  */
/* Copyright (c) Genus Microprogramming, Inc. 1988-89  All Rights Reserved. */
/****************************************************************************
  Function declarations for the Super VGA Library, for C.
  Microsoft C version 5.1                Programmer: Chris Howard
****************************************************************************/

/* Display modes */
#define svTEXT      0                   /* Text mode                    */
#define svGRAPHICS  1                   /* Graphics mode                */

/* Display types */
#define svMINDISP   0
#define svDISP_2D   0                   /* Tseng    2DH (640x350x256)   */
#define svDISP_2E   1                   /* Tseng    2EH (640x480x256)   */
#define svDISP_30   2                   /* Tseng    30H (800x600x256)   */
#define svDISP_5E   3                   /* Paradise 5EH (640x400x256)   */
#define svDISP_5F   4                   /* Paradise 5FH (640x480x256)   */
#define svDISP_66   5                   /* Video 7  66H (640x400x256)   */
#define svDISP_67   6                   /* Video 7  67H (640x480x256)   */
#define svDISP_69   7                   /* Video 7  69H (800x600x256)   */
#define svMAXDISP   7

/* Logical Operations */
#define svSET       0                   /* SET pixel value directly     */
#define svAND       1                   /* AND pixel value with data    */
#define svOR        2                   /* OR  pixel value with data    */
#define svXOR       3                   /* XOR pixel value with data    */

/* Chip sets */
#define svUNKNOWN   0                   /* Unknown chip set             */
#define svTSENG     1                   /* Tseng Labs                   */
#define svPARA      2                   /* Paradise                     */
#define svV7        3                   /* Video 7                      */

/* Error Codes */
#define svSUCCESS   0                   /* Successful                   */
#define svBADMODE   -1                  /* Bad display mode             */

/* Functions */
extern  int   far pascal  svSetDisplay      (int);
extern  int   far pascal  svSetMode         (int);
extern  int   far pascal  svSetOp           (int);
extern  int   far pascal  svPutPixel        (int,int,int);
extern  int   far pascal  svQueryChipset    (void);
```

End Listing Six

Listing Seven

```
/* svTest.C                                                                 */
/* Copyright (c) Genus Microprogramming, Inc. 1988-89  All Rights Reserved. */
/****************************************************************************
  This is a simple test program, for testing the Super VGA QueryChipset
  and PutPixel functions.
  Compile: CL /c /AS svTest.C
           link svTest,,,svLib;
  Microsoft C version 5.1                Programmer: Chris Howard
```

```
*****************************************************************************/

#include <stdio.h>
#include <conio.h>
#include "svlib.h"

/* Global data */
static char *chip[]    = {"[Unknown]","Tseng Labs","Paradise","Video7");

/**********/

/* This is a crude box drawing routine. Uses the svPutPixel function to draw*/
void svPutSquare(x,y,w,c)
int  x,y,w,c;

{
  int  i;
  for (i=0; i<w; i++) {
    svPutPixel(x+i,y,  c);
    svPutPixel(x+i,y+w,c);
    svPutPixel(x,  y+i,c);
    svPutPixel(x+w,y+i,c);
  }

} /* end of svPutSquare */

/**********/
void main(void)
{
  int  i,j,k,chipset,svdisplay,retcode;

  /* Display a header */
  printf("\n\nSuper VGA Test Program\n\n");

  /* Query the chipset, and see what we have */
  chipset = svQueryChipset();

  printf("Your Super VGA chipset is: %s\n\n",chip[chipset]);
  printf("Press any key to continue ...\n\n");
  getch();

  /* If we have a chipset we recognize, keep going */
  if (chipset != svUNKNOWN) {

    /* Based on the chipset, select a display type for 256 colors */
    switch (chipset) {
      case svTSENG:
        /* 640x480x256 */
        svdisplay = svDISP_2E;
        break;
      case svPARA:
        /* 640x400x256 */
        svdisplay = svDISP_5E;
        break;
      case svV7:
        /* 640x480x256 */
        svdisplay = svDISP_67;
        break;
    }

    /* Set the display and mode */
    retcode = svSetDisplay(svdisplay);
    retcode = svSetMode(svGRAPHICS);

    /* If the mode was set successfully, try displaying some pixels */
    if (retcode == svSUCCESS) {

      /* Display a rainbow bar, a few lines thick */
      for (j=0; j<10; j++) {
        for (i=0; i<256; i++)
          svPutPixel(200+i,200+j,i);
      }

      /* Demonstrate logical operations by XORing a 'square' across rainbow */
      retcode = svSetOp(svXOR);
      for (i=0; i<256; i++) {
        svPutSquare(200+i,200,10,15);
        /* Dummy delay */
        for (j=0; j<5000; j++)
          k = j;
        svPutSquare(200+i,200,10,15);
      }

      /* Wait for a key */
      getch();

      /* Return to text mode, and display completion message */
      svSetMode(svTEXT);

      printf("svTest completed\n\n");
    }
  }
  else {
    /* We could not recognize the chip, so suggest something */
    printf("No test can be run ...\n\n");
    printf("If you are sure of your chipset, try changing the program so\n");
    printf("the chip type is forced.\n\n");
  }
} /* end of main */
```

End Listing Seven

Listing Eight

```
# svLIB Make File
#
# Copyright (c) Genus Microprogramming, Inc. 1988-89 All Rights Reserved.    #
```

(Listing continued on page 92.)

Listing Eight *(Listing continued, text begins on page 16.)*

```
#############################################################################
# This make file is for making the Super VGA Library.                       #
# Usage: Make svLib /I                                                      #
# Microsoft ASM 5.1                          Programmer: Chris Howard       #
#############################################################################

# Compiler and linker flags
AFLAGS   = /DLINT_ARGS /W2 /B63 /ZI
#          /D  = Define          /W2 = Max ASM warnings  /B  = Buffer size
CFLAGS   = /G0 /AS /Os /c /Zi
#          /G0 = 8088 code       /AS = Small model       /Os = Optimize Size
#          /c  = Compile only
DFLAGS   = /DLINT_ARGS /W3
#          /D  = Define          /W3 = Max C warnings
LFLAGS   =

# Compiler Programs
CC       = cl   $(CFLAGS)  $(DFLAGS)
ASM      = masm $(AFLAGS)
LINK     = link $(LFLAGS)
LIB      = lib

# ASM Include files
SVDEFS   = svDefs.inc
SVMACS   = svMacs.inc

# Libraries
SVLIB    = svLib

# Remember:
#
#   $*   = Base name of the outfile (without extension)
#   $@   = Complete outfile name
#   $**  = Complete list of infiles
#

############

# Query Chipset
svQC.obj: $*.asm $(SVDEFS) $(SVMACS)
        $(ASM) $*,$@;
        $(LIB) $(SVLIB) -$* +$@;

# Pixel Addressing
svPA.obj: $*.asm $(SVDEFS) $(SVMACS)
        $(ASM) $*,$@;
        $(LIB) $(SVLIB) -$* +$@;

# Put Pixel
svPP.obj: $*.asm $(SVDEFS) $(SVMACS)
        $(ASM) $*,$@;
        $(LIB) $(SVLIB) -$* +$@;

# End.
```

End Listing Eight

Listing Nine

```
# svTest Make File #
# Copyright (c) Genus Microprogramming, Inc. 1988-89 All Rights Reserved.   #
#############################################################################
# This make file is for making the Super VGA Test program.                  #
# Microsoft C 5.1                            Programmer: Chris Howard       #
#############################################################################

# Compiler and linker flags
CFLAGS   = /G0 /AS /Os /c
#          /G0 = 8088 code       /AS = Small model       /Os = Optimize Size
#          /c  = Compile only
DFLAGS   = /DLINT_ARGS /W3 /Zi
#          /D  = Define          /W3 = Max C warnings    /Zi = Codeview
LFLAGS   = /
#          /CO = Codeview

# Compiler Programs
CC       = cl   $(CFLAGS) $(DFLAGS)
ASM      = masm
LINK     = link $(LFLAGS)

# Include files
SV       = svlib.h

# Libraries
SVLIB    = svlib.lib

############

# The Test program
svtest.obj: $*.c  $(SV)
        $(CC) $*.c

# Now link it all together
svtest.exe: $*.obj $(SVLIB)
        $(LINK) $*,,,$(SVLIB);
```

End Listings

Listing One *(Text begins on page 30.)*

```
#include <graph.h>
#include <math.h>

int xbase, ybase;
unsigned xAspect, yAspect;

void SetAspect(double aspect)
{
   xAspect = 0;
   yAspect = 0;
   aspect = fabs(aspect);
   if (aspect != 1.0)
      if (aspect > 1.0)
         yAspect = 65536.0 / aspect;
      else
         xAspect = 65536.0 * aspect;
}

void plot(int x, int y)
{
   if (xAspect == 0)
      if (yAspect == 0)
         _setpixel(x+xbase, y+ybase);
      else
         _setpixel(x+xbase, ybase + (((long) y * (long) yAspect) >> 16));
   else
      _setpixel(xbase + (((long) x * (long) xAspect) >> 16), y+ybase);
}

void plot8(int x, int y)
{
   plot(x,y);
   plot(-x,y);
   plot(x,-y);
   plot(-x,-y);
   plot(y,x);
   plot(-y,x);
   plot(y,-x);
   plot(-y,-x);
}

void circle(int radius)
{
   int x,y,sum;

   x = 0;
   y = radius << 1;
   sum = 0;

   while (x <= y)
   {
      if ( !(x & 1) )   /* plot if x is even */
         plot8( x >> 1, (y+1) >> 1);
      sum += (x << 1) + 1;
      x++;
      if (sum > 0)
      {
         sum -= (y << 1) - 1;
         y--;
      }
   }
}

int main()
{
   _setvideomode(_VRES2COLOR);
   xbase = 320;
   ybase = 240;
   SetAspect(1.0);
   circle(100);
   SetAspect(2.0);
   circle(100);
   SetAspect(0.5);
   circle(100);
}
```

End Listing

Listing One *(Text begins on page 36.)*

```
/* Created by Victor J. Duvanenko.    Straight forward implementation of the
Cohen-Sutherland line clipping algorithm. */

#define BOOLEAN     int
#define TRUE          1
#define FALSE         0
#define OK            0
#define NOT_OK       -1
#define ACCEPT        TRUE
#define REJECT        FALSE

/* Clipping rectangle boundaries - accessable to all routines */
extern double
            y_bottom,
            y_top,
            x_right,
            x_left,
            z_front,
            z_back;

*/----------------------- Cohen-Sutherland 2D algorithm --------------------
  Procedure that sets four bits, and outcode, to designate a region of
  point existance relative to the clipping rectangle.  For a full explanation
  see Foley and Van Dam p.146 Fig. 4.5
  Bit 0 - point is above       window
  Bit 1 - point is below       window
  Bit 2 - point is to right of window
  Bit 3 - point is to left  of window
  The algorithm is border inclusive - border is included in the window.
------------------------------------------------------------------------*/
static void outcodes( x, y, outcode )
double  x, y;
int  *outcode;
{
    *outcode = 0;
    if ( y > y_top    )    *outcode |= 1;
    if ( y < y_bottom )    *outcode |= 2;
    if ( x > x_right  )    *outcode |= 4;
    if ( x < x_left   )    *outcode |= 8;
}
/*------------------------------------------------------------------------
  Procedure that checks for trivial rejects - if both end points are above,
  below, to the right, or to the left of the window.
  This procedure can be converted into a macro, for performance improvement.
------------------------------------------------------------------------*/
static reject_check( outcode1, outcode2 )
int  outcode1, outcode2;
{
    if ( outcode1 & outcode2 )
        return( TRUE );
    return( FALSE );
}
/*------------------------------------------------------------------------
  Procedure that checks for trivial accept - if both end points are within
  the window. This procedure can also be replaced by a macro, for speed.
------------------------------------------------------------------------*/
static accept_check( outcode1, outcode2 )
int  outcode1, outcode2;
{
    if (( !outcode1 ) && ( !outcode2 ))
        return( TRUE );
    return( FALSE );
}
/*------------------------------------------------------------------------
  Procedure that exchanges the endpoints and their outcodes.
------------------------------------------------------------------------*/
static void swap_pts( x0, y0, outcode0, x1, y1, outcode1 )
double  *x0, *y0;
int  *outcode0;
double  *x1, *y1;
int  *outcode1;
{
    double  tmp;
    int     tmp_i;

    tmp = *x0;      /* exchange the x's */
    *x0 = *x1;
    *x1 = tmp;
    tmp = *y0;      /* exchange the y's */
    *y0 = *y1;
    *y1 = tmp;
    tmp_i     = *outcode0;      /* exchange the outcodes */
    *outcode0 = *outcode1;
    *outcode1 = tmp_i;
}
/*------------------------------------------------------------------------
  Procedure that clips in 2D using Cohen-Sutherland algorithm.  The algorithm
  has been modified to separate the drawing function from clipping.  This way
  clipping is one of the pipelined processes.  This procedure lets the calling
  routine know whether the line has been accepted or rejected, so that the
  caller knows whether to draw it, with possibly modified points, or not.
------------------------------------------------------------------------*/
clip_2d_cs( x0, y0, x1, y1 )
double  *x0, *y0;       /* first  end point */
double  *x1, *y1;       /* second end point */
{
    int      outcode0, outcode1;
    /* Adjust end points until it's possible to trivially accept or reject */
    while( TRUE )                   /* one adjustment per iteration */
    {
        outcodes( *x0, *y0, &outcode0 );
        outcodes( *x1, *y1, &outcode1 );
        if ( reject_check( outcode0, outcode1 ))
            return( REJECT );
        if ( accept_check( outcode0, outcode1 ))
            return( ACCEPT );

        /* subdivide line since at most one endpoint is inside       */
        /* First, if P0 is inside window, exchange points P0 and P1  */
        /* and their outcodes to guarantee that P0 is outside window */
        if ( !outcode0 )
            swap_pts( x0, y0, &outcode0, x1, y1, &outcode1 );

        /* Now perform a subdivision, move P0 to the intersection point.
           use the formulas y = y1 +   slope   * (x - x1)
                            x = x1 + (1/slope) * (y - y1)
           Note that we don't have to worry about division by 0 in any of
           these cases.  If a line is horizontal, then if any of its end
           points were above the window, it would have been trivially rejected.
           So, the only case possible is that the end points are left or
           right of the window (outcode0 & (4 or 8)).                    */

        if ( outcode0 & 1 )
        {                           /* divide line at top of window */
            *x0 += ( *x1 - *x0 ) * ( y_top - *y0 ) / ( *y1 - *y0 );
            *y0  = y_top;
        }
        else if ( outcode0 & 2 )
        {                           /* divide line at bottom of window */
            *x0 += ( *x1 - *x0 ) * ( y_bottom - *y0 ) / ( *y1 - *y0 );
            *y0  = y_bottom;
        }
        else if ( outcode0 & 4 )
        {                           /* divide line at right edge of window */
            *y0 += ( *y1 - *y0 ) * ( x_right  - *x0 ) / ( *x1 - *x0 );
            *x0  = x_right;
        }
        else if ( outcode0 & 8 )
        {                           /* divide line at left edge of window */

           *y0 += ( *y1 - *y0 ) * ( x_left  - *x0 ) / ( *x1 - *x0 );
            *x0  = x_left;
        }
    }
}
/* ------------------------ END of 2D CS algorithm ------------------------ */
```

End Listing One

Listing Two

```
/* Created by Victor J. Duvanenko  An improved implementation of the
   Cohen-Sutherland line clipping algorithm. */

#define BOOLEAN     int
#define TRUE          1
#define FALSE         0
#define OK            0
#define NOT_OK       -1
#define ACCEPT        TRUE
#define REJECT        FALSE

/* Clipping rectangle boundaries - accessable to all routines */
extern double
            y_bottom,
            y_top,
            x_right,
            x_left,
            z_front,
            z_back;

/*------------- Cohen-Sutherland-Duvanenko 2D and 3D algorithms ---------- */
  Procedure that sets four bits, and outcode, to designate a region of
  point existance relative to the clipping rectangle.  For a full explanation
  see Foley and Van Dam p.146 Fig. 4.5
  Bit 0 - point is above       window
  Bit 1 - point is below       window
  Bit 2 - point is to right of window
  Bit 3 - point is to left  of window
  The algorithm is border inclusive - border is included in the window.
  Defining the procedure as a macro removes the need to pass the data on
  the stack - and gains additional speed.
  Note, that top and bottom (and right and left) are mutually exclusive
  areas.  That's the reason for else's - reduces compares 4 down to 3.
------------------------------------------------------------------------*/
#define OUTCODES_CSD( X, Y, OUTCODE )           \
    ( OUTCODE ) = 0;                            \
    if ((Y) > y_top    )          OUTCODE |= 1; \
    else  if ((Y) < y_bottom )    OUTCODE |= 2; \
    if ((X) > x_right  )          OUTCODE |= 4; \
    else  if ((X) < x_left   )    OUTCODE |= 8;
/*------------------------------------------------------------------------
  Procedure that exchanges the endpoints and their outcodes.
  Defining the procedure as a macro removes the need to pass the data on
  the stack - and gains additional speed.
------------------------------------------------------------------------*/
#define SWAP_PTS_CSD( X0, Y0, OUTCODE0, X1, Y1, OUTCODE1 )   \
    tmp = *X0;        /* exchange the x's */                 \
    *X0 = *X1;                                               \
    *X1 = tmp;                                               \
    tmp = *Y0;        /* exchange the y's */                 \
    *Y0 = *Y1;                                               \
    *Y1 = tmp;                                               \
    tmp_i     = OUTCODE0;      /* exchange the outcodes */   \
    OUTCODE0 = OUTCODE1;                                     \
    OUTCODE1 = tmp_i;
/*------------------------------------------------------------------------
  Procedure that clips in 2D using Cohen-Sutherland algorithm.  The algorithm
  has been modified to separate the drawing function from clipping.  This way
  clipping is one of the pipelined processes.  This procedure lets the calling
  routine know whether the line has been accepted or rejected, so that the
  caller knows whether to draw it, with possibly modified points, or not.
------------------------------------------------------------------------*/
clip_2d_csd( x0, y0, x1, y1 )
double  *x0, *y0;       /* first  end point */
double  *x1, *y1;       /* second end point */
{
    register  unsigned  outcode0, outcode1;
```

(Listing continued on page 100)

Listing Two *(Listing continued, text begins on page 36.)*

```
double   dx, dy;          /* change in x and change in y   */
BOOLEAN  intersect;
double  tmp;              /* needed for the SWAP procedure */
int     tmp_i;            /* needed for the SWAP procedure */

OUTCODES_CSD( *x0, *y0, outcode0 );
OUTCODES_CSD( *x1, *y1, outcode1 );
if ( outcode0 & outcode1 )          /* trivial reject */
    return( REJECT );
if ( ! ( outcode0 | outcode1 ))     /* trivial accept */
    return( ACCEPT );

/* Calculate the subproducts of the slope only once. */
/* These must be calculated in all cases. */
dx = *x1 - *x0;
dy = *y1 - *y0;
intersect = FALSE;

/* Adjust end points until it's possible to trivially accept or reject */
while( TRUE )                   /* one adjustment per iteration */
{
    /* subdivide line since at most one endpoint is inside       */
    /* First, if P0 is inside window, exchange points P0 and P1  */
    /* and their outcodes to guarantee that P0 is outside window */
    if ( !outcode0 )
    {
        SWAP_PTS_CSD( x0, y0, outcode0, x1, y1, outcode1 );
        intersect = TRUE;         /* the line intersects the window */
    }

    /* Now perform a subdivision, move P0 to the intersection point.
       use the formulas y = y1 +   slope   * (x - x1)
                        x = x1 + (1/slope) * (y - y1)
       Note that we don't have to worry about division by 0 in any of
       these cases.  If a line is horizontal, then if any of its end
       points were above the window, it would have been trivially rejected.
       So, the only case possible is that the end points are left or
       right of the window (outcode0 & (4 or 8)).                    */

    if ( outcode0 & 1 )
    {                       /* divide line at top of window */
        *x0 += dx * ( y_top - *y0 ) / dy;
        *y0  = y_top;
        outcode0 = 0;
        if ( *x0 > x_right  )          outcode0 |= 4;
        else  if ( *x0 < x_left   )    outcode0 |= 8;
    }
    else if ( outcode0 & 2 )
    {                       /* divide line at bottom of window */
        *x0 += dx * ( y_bottom - *y0 ) / dy;
        *y0  = y_bottom;
        outcode0 = 0;
        if ( *x0 > x_right  )          outcode0 |= 4;
        else  if ( *x0 < x_left   )    outcode0 |= 8;
    }
    else if ( outcode0 & 4 )
    {                       /* divide line at right edge of window */
        *y0 += dy * ( x_right  - *x0 ) / dx;
        *x0  = x_right;
        outcode0 = 0;
        if ( *y0 > y_top    )          outcode0 |= 1;
        else  if ( *y0 < y_bottom )    outcode0 |= 2;
    }
    else if ( outcode0 & 8 )
    {                       /* divide line at left edge of window */

       *y0 += dy * ( x_left  - *x0 ) / dx;
        *x0  = x_left;
        outcode0 = 0;
        if ( *y0 > y_top    )          outcode0 |= 1;
        else  if ( *y0 < y_bottom )    outcode0 |= 2;
    }
    if ( outcode0 & outcode1 )          /* trivial reject */
        return( REJECT );
    if ( ! ( outcode0 | outcode1 ))     /* trivial accept */
        return( ACCEPT );
    }
}
/* ---------------------- END of 2D CSD algorithm ---------------------- */
```

End Listing Two

Listing Three

```
/* Created by Victor J. Duvanenko  Program that generates random 3D points,
   that will be used for testing performance of line clipping algorithms. */

#include <stdio.h>
#include <stdlib.h>

#define BOOLEAN     int
#define TRUE          1
#define FALSE         0
#define OK            0
#define NOT_OK       -1
#define NUM_LINES   1000       /* generate this many lines */
#define ROUND(x)    ((x) > 0.0  ?  (int)((x) + 0.5) : (int)((x) - 0.5))
#define DEBUG     FALSE        /* en(dis)able debug sections */

/*-------------------------------------------------------------------------
  Procedure that generates any number of 3D points - randomly.
-------------------------------------------------------------------------*/
generate_vertex_data( num_points )
{
    register  i;
    double  x, y, z;
    char *fname;
    FILE *fp;

    /* Fix the file name, since performance benchmarks are required. */
    /* Therefore, user interaction should be completely eliminated.  */
    fname = "clip1.dat";

    if (( fp = fopen( fname, "w" )) == NULL )
    {
        printf("Couldn't open file %s.\n", fname );
        return( TRUE );
    }
    for( i = 0; i < num_points; i++ )
    {
        /* center the values around 0.0, so as to have an equal number */
        /* of negative and positive values.                            */
        x = (double)rand() / (double)RAND_MAX * 64000.0 - 32000.0;
        y = (double)rand() / (double)RAND_MAX * 64000.0 - 32000.0;
        z = (double)rand() / (double)RAND_MAX * 64000.0 - 32000.0;
        fprintf( fp, "%12.2lf  %12.2lf  %12.2lf\n", x, y, z );
    }
    fclose( fp );
}
/*------------------------- MAIN -----------------------------------*/
main( argc, argv )
int argc;
char **argv;
{
    generate_vertex_data( NUM_LINES << 1 );
    return( 0 );                    /* everything went fine */
}
```

End Listings

Listing One *(Text begins on page 46.)*

```
/*-------------------------------------------------------------
  BEZIER.C An application which draws characters (or anything else)
  defined as a collection of Bezier curves. As is this programs
  works fine on all displays except CGA in the 320x200 mode since
  it expects a screen that is 640x200. To also use the deCasteljau method.
-------------------------------------------------------------*/

#include <graphics.h>
#include <conio.h>

/* Definitions for defining fonts */
#define POINT_DEF       1000.0  /* The point size fonts are defined in */
#define BFLAG   -POINT_DEF - 1  /* A Special flag used as a sentinal */
#define BNULL   0.0             /* For readability */
float DPU =  600.0 / POINT_DEF; /* The step between pixels */

/* Paramters for controlling the deCasteljau rendering method */
int B_cutoff = 3;
int B_level  = 0;

/* Structures for defining Bezier bounding boxes */
typedef struct {
  float x, y;
} XY;

typedef struct {
  XY a, b, c, d;
} BEZIER_BOX;

/* Do a little letter doodling */
main(argc, argv)
int argc;
char *argv[];
{
  int draw_bezier1();
  int draw_bezier2();

  int (*draw_fun)();
  char label[80];
  int gdriver = DETECT, gmode;
  BEZIER_BOX lower_a[] = {
    { {   0.0,  591.0}, {   0.0,  864.0}, {1000.0,  864.0}, {1000.0,  591.0} },
    { {1000.0,  591.0}, {1000.0,  591.0}, {1000.0,   45.0}, {1000.0,   45.0} },
    { {1000.0,   45.0}, {1000.0,   45.0}, { 727.0,   45.0}, { 727.0,   45.0} },
    { { 727.0,   45.0}, { 727.0,   45.0}, { 727.0,  136.0}, { 727.0,  136.0} },
    { { 727.0,  136.0}, { 727.0,    0.0}, {   0.0,    0.0}, {   0.0,  227.0} },
    { {   0.0,  227.0}, {   0.0,  545.0}, { 727.0,  545.0}, { 727.0,  364.0} },
    { { 727.0,  364.0}, { 727.0,  500.0}, { 727.0,  500.0}, { 727.0,  545.0} },
    { { 727.0,  545.0}, { 727.0,  727.0}, { 272.0,  727.0}, { 272.0,  592.0} },
    { { 272.0,  592.0}, { 272.0,  592.0}, {   0.0,  592.0}, {   0.0,  592.0} },
    { { 636.0,  280.0}, { 636.0,   90.0}, { 136.0,  119.0}, { 136.0,  250.0} },
    { { 136.0,  250.0}, { 136.0,  437.0}, { 636.0,  437.0}, { 636.0,  280.0} },
    { { BFLAG,  BNULL}, { 909.0,  228.0}, { BNULL,  BNULL}, { BNULL,  BNULL} }
  };

  if(argc != 2) {
    print_usage();
  }

/* Check options */
  switch(argv[1][0]) {
    case 'd':   /* deCasteljau rendering */
    case 'D':
      strcpy(label, "deCasteljau rendering:");
      draw_fun = draw_bezier2;
      break;
    case 'l':   /* Literal rendering */
    case 'L':
      strcpy(label, "Literal rendering:");
      draw_fun = draw_bezier1;
      break;
    default:
      printf("Unknown option: %c\n", argv[1][0]);
      print_usage();
      break;
  }

  /* NOTE: the following pathname may need to be modified for your system. */
  initgraph(&gdriver, &gmode, "C:\\turboc");
   if(gdriver < 0 ) {
    printf("Unable to initialize graphics, error code: %d", gdriver);
    exit(0);
  }

/* Now draw the letters with both rendering methods */
  outtextxy(1, 1, label);

  draw_letter(30.0, 160.0, lower_a, 180.0,
     WHITE, SLASH_FILL, WHITE, draw_fun);
  draw_letter(220.0, 160.0, lower_a, 120.0,
     WHITE, SOLID_FILL, WHITE, draw_fun);
  draw_letter(350.0, 160.0, lower_a, 90.0,
     BLUE, LINE_FILL, BLUE, draw_fun);
  draw_letter(450.0, 160.0, lower_a, 40.0,
     WHITE, SOLID_FILL, WHITE, draw_fun);
  draw_letter(500.0, 160.0, lower_a, 20.0,
     WHITE, LINE_FILL, WHITE, draw_fun);
  draw_letter(530.0, 160.0, lower_a, 10.0,

      WHITE, SOLID_FILL, WHITE, draw_fun);

  outtextxy(1, 190, "Press any key to continue...");
  getch();
  closegraph();
}

/*-------------------------------------------------
  Draws a letter at a given location of a given size. The
  outline color, fill color and fill style are also specified.
-------------------------------------------------*/
draw_letter(at_x, at_y, letter, size,
  outline_color, fill_style, fill_color, draw_bezier)
float at_x, at_y;
BEZIER_BOX letter[];
float size;
int outline_color;
int fill_style;
int fill_color;
int draw_bezier();
{
  int i = 0;
  BEZIER_BOX bline;
  float scale;

  scale = size / POINT_DEF;

  setcolor(outline_color);
  setfillstyle(fill_style, fill_color);
  setlinestyle(SOLID_LINE, 0, NORM_WIDTH);

  while(letter[i].a.x != BFLAG) {
    bline.a.x = at_x + (letter[i].a.x * scale);
    bline.a.y = at_y - (letter[i].a.y * scale);
    bline.b.x = at_x + (letter[i].b.x * scale);
    bline.b.y = at_y - (letter[i].b.y * scale);
    bline.c.x = at_x + (letter[i].c.x * scale);
    bline.c.y = at_y - (letter[i].c.y * scale);
    bline.d.x = at_x + (letter[i].d.x * scale);
    bline.d.y = at_y - (letter[i].d.y * scale);
    draw_bezier(&bline);
    i++;
  }

  floodfill((int)(at_x + letter[i].b.x * scale),
            (int)(at_y - letter[i].b.y * scale),
             fill_color);

}

/*-----------------------------------------------
  Draws a bezier curve as defined by the passed Bezier bounding box.
-----------------------------------------------*/
draw_bezier1(bcurve)
BEZIER_BOX *bcurve;
{
  float x, y;
  float tm;
  float t;

  moveto((int)bcurve->a.x, (int)bcurve->a.y);

  for(t = 0.0; t <= 1.0; t += 0.01) {
    x = (1 - t) * (1 - t) * (1 - t) * bcurve->a.x
      + 3 * t * (t - 1) * (t - 1) * bcurve->b.x
      + 3 * t * t * (1 - t) * bcurve->c.x
      + t * t * t * bcurve->d.x;
    y = (1 - t) * (1 - t) * (1 - t) * bcurve->a.y
      + 3 * t * (t - 1) * (t - 1) * bcurve->b.y
      + 3 * t * t * (1 - t) * bcurve->c.y
      + t * t * t * bcurve->d.y;
    lineto((int)x, (int)y);
  }
}

/*-------------------------------------------------------
  Draws a bezier curve as a series of line segments.
  This uses the deCasteljau algorithm.
-------------------------------------------------------*/
draw_bezier2(bcurve)
BEZIER_BOX *bcurve;
{
  extern int B_level;

  BEZIER_BOX b_box;
  XY a0, d0;
  XY a1, a2, a3;
  XY b1, b2;
  XY c1;

  if(accurate(bcurve) == 1) {
    draw_hull(bcurve);
    return(0);
  }
  B_level++;

  a0.x = bcurve->a.x;
  a0.y = bcurve->a.y;

  d0.x = bcurve->d.x;
  d0.y = bcurve->d.y;
  a1.x = a0.x + (bcurve->b.x - a0.x)/2;
  a1.y = a0.y + (bcurve->b.y - a0.y)/2;

  b1.x = bcurve->b.x + (bcurve->c.x - bcurve->b.x)/2;
  b1.y = bcurve->b.y + (bcurve->c.y - bcurve->b.y)/2;

  c1.x = bcurve->c.x + (bcurve->d.x - bcurve->c.x)/2;
  c1.y = bcurve->c.y + (bcurve->d.y - bcurve->c.y)/2;

  a2.x = a1.x + (b1.x - a1.x)/2;
  a2.y = a1.y + (b1.y - a1.y)/2;

  b2.x = b1.x + (c1.x - b1.x)/2;
  b2.y = b1.y + (c1.y - b1.y)/2;

  a3.x = a2.x + (b2.x - a2.x)/2;
  a3.y = a2.y + (b2.y - a2.y)/2;
```

(Listing continued on page 104.)

Listing One *(Listing continued, text begins on page 46.)*

```
  b_box.a.x = a0.x;
  b_box.a.y = a0.y;
  b_box.b.x = a1.x;
  b_box.b.y = a1.y;
  b_box.c.x = a2.x;
  b_box.c.y = a2.y;
  b_box.d.x = a3.x;
  b_box.d.y = a3.y;
  draw_bezier2(&b_box);

  b_box.a.x = a3.x;
  b_box.a.y = a3.y;
  b_box.b.x = b2.x;
  b_box.b.y = b2.y;
  b_box.c.x = c1.x;
  b_box.c.y = c1.y;
  b_box.d.x = d0.x;
  b_box.d.y = d0.y;
  draw_bezier2(&b_box);

  B_level--;
  return(0);
}

/*-----------------------------------------------
  Determines if the bezier curve has been determined with enough accuracy.
  Returns 1 if the passed Bezier bounding box is accurate enough, 0 otherwise.
-----------------------------------------------*/
accurate(b_box)
BEZIER_BOX *b_box;
{
  extern int B_level;
  extern int B_cutoff;

  if(B_level >= B_cutoff) {
    return(1);
  } else {
    return(0);
  }
}

/*-----------------------------------------------
  Draws a hull as defined by a Bezier bounding box.
-----------------------------------------------*/
draw_hull(b_box)
BEZIER_BOX *b_box;
{
  moveto((int) b_box->a.x, (int) b_box->a.y);
  lineto((int) b_box->b.x, (int) b_box->b.y);
  lineto((int) b_box->c.x, (int) b_box->c.y);
  lineto((int) b_box->d.x, (int) b_box->d.y);
}

/*-----------------------------------------------
  Prints the usage message then exits.
-----------------------------------------------*/
print_usage()
{
  printf("Prints a series of lower case 'a's in decreasing size. You may\n");
  printf("select between deCasteljau or literal rendering methods.\n");
  printf("Proper usage:\n");
  printf("\n");
  printf("    bezier d|l\n");
  printf("\n");
  printf("If 'd' option is specified the deCasteljau rendering method\n");
  printf("is used. If 'l' option is specified a literal Bezier rendering\n");
  printf("method is used.\n");
  exit(0);
}
```

End Listing

Listing One *(Text begins on page 54.)*

```
View subclass: #BarView
        instanceVariableNames: 'maximumValue '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Demo-Counter'!

!BarView methodsFor: 'accessing'!

barFrame
        ^self insetDisplayBox insetBy: (50 @ 10 corner: 10 @ 10)!
labelCount
        ^5!
maximumValue
        ^maximumValue!
maximumValue: anInteger
        maximumValue _ anInteger!
positionFor: value
        ^self barFrame height * value / self maximumValue! !
!BarView methodsFor: 'displaying'!
clearBar
        | height bar corner |

        corner _ self barFrame bottomLeft.
        height _ self positionFor: self model value + 1.
        height _ height min: (self insetDisplayBox height-10).
        bar _ corner - (0 @ height) extent: self barFrame width @ height.
        Display white: bar.!
displayBar
        | height bar corner |
        corner _ self barFrame bottomLeft.
        height _ self positionFor: self model value.
        height _ height min: (self insetDisplayBox height-10).
        bar _ corner - (0 @ height) extent: self barFrame width @ height.
        Display black: bar.
        Display fill: (bar insetBy: 2 @ 2)
                mask: Form darkGray.!
displayView
        self displayYLabels.
        self displayBar!
displayYLabels
        | count label increment |
        count _ self labelCount.
        increment _ self maximumValue / count.
        label _ 0.
        (count+1) timesRepeat:
                [label printString displayAt: (self barFrame bottomLeft -
                                        (35 @ ((self positionFor: label)+8))).
                 label _ label + increment]!
update: aParameter
        self clearBar.
        self displayBar.! !
!BarView methodsFor: 'controller access'!

defaultControllerClass
        "Answer the class of a typically useful controller."

        ^CounterController! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

BarView class
        instanceVariableNames: ''!

!BarView class methodsFor: 'instance creation'!

newWithHeight: aNumber
        "Create a new BarView for displaying value between 0 and aNumber"
        ^super new maximumValue: aNumber!

open: aModel
        "Open a view for a new counter."
        "BarView open: Counter new."

        | aBarView topView |

        aBarView _ (BarView newWithHeight: 10) model: aModel.
        aBarView borderWidth: 2.
        aBarView insideColor: Form white.

        topView _ StandardSystemView new label: 'Counter'.
        topView minimumSize: 80@40.
        topView addSubView: aBarView.

        topView controller open!

open: aModel withHeight: anInteger
        "Open a view for a new counter."

        "BarView open: Counter new withHeight: 20."

        | aBarView topView |

        aBarView _ (BarView newWithHeight: anInteger) model: aModel.
        aBarView borderWidth: 2.
        aBarView insideColor: Form white.

        topView _ StandardSystemView new label: 'Counter'.
        topView minimumSize: 80@40.
        topView addSubView: aBarView.

        topView controller open! !

View subclass: #CounterView
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Demo-Counter'!

!CounterView methodsFor: 'displaying'!
displayView
        "Display the value of the model."
        | box pos |

        "Position the text at the left side of the display area."
        box _ self insetDisplayBox.            "get the view's box"
        pos _ box origin + (4 @ (box extent y / 3)).
                                "put the text 1/3 of the way down to the left"
        "Concatenate the components of the output string and display them."
        ('val: ', self model value asInteger printString, ' ')
                                                asDisplayText displayAt: pos.! !
!CounterView methodsFor: 'updating'!
update: aParameter
        "Simply redisplay everything."
        self display! !

!CounterView methodsFor: 'controller access'!
defaultControllerClass
        "Answer the class of a typically useful controller."

        ^CounterController! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

CounterView class
        instanceVariableNames: ''!

!CounterView class methodsFor: 'instance creation'!
open: aModel
        "Open a view for a new counter."

        "CounterView open: Counter new."

        | aCounterView topView |

        aCounterView _ CounterView new model: aModel.
        aCounterView borderWidth: 2.
        aCounterView insideColor: Form white.

        topView _ StandardSystemView new label: 'Counter'.
        topView minimumSize: 80@40.
        topView addSubView: aCounterView.

        topView controller open!

openWithGraphicalButtons: aModel
        "Open a view for a new counter that has fixed graphical buttons for
                                        incrementing and decrementing the value."
        "CounterView openWithGraphicalButtons: Counter new"

        | aCounterView topView switch aSwitchView |

        aCounterView _ CounterView new model: aModel.
        aCounterView insideColor: Form white.

        topView _ StandardSystemView new label: 'Counter'.
        topView minimumSize: 120 @ 80.
        topView maximumSize: 400 @ 300.                                                 "add
main view"
        topView addSubView: aCounterView
                in: (0.4 @ 0 extent: 0.6 @ 1)
                borderWidth: (0 @ 2 extent: 2 @ 1).

        switch _ Button newOff.                 "add increment button"
                switch onAction: [aCounterView model increment].
                aSwitchView _ SwitchView new model: switch.
                aSwitchView label: ('+' asDisplayText form magnifyBy: 2@2).
                aSwitchView insideColor: Form lightGray.
        topView addSubView: aSwitchView
                in: (0 @ 0 extent: 0.4 @ 0.5)
                borderWidth: (2 @ 2 extent: 0 @ 0).

        switch _ Button newOff.         "add decrement button"
                switch onAction: [aCounterView model decrement].
                aSwitchView _ SwitchView new model: switch.
                aSwitchView label: ('-' asDisplayText form magnifyBy: 2@2).
                aSwitchView insideColor: Form lightGray.
        topView addSubView: aSwitchView
                in: (0 @ 0.5 extent: 0.4 @ 0.5)
                borderWidth: (2 @ 1 extent: 0 @ 2).
        topView controller open!

openWithTextButtons: aModel
        "Open a view for a new counter that has fixed text buttons for
                                        incrementing and decrementing the value."
        "CounterView openWithTextButtons: Counter new"

        | aCounterView topView switch aSwitchView |

        aCounterView _ CounterView new model: aModel.
        aCounterView insideColor: Form white.

        topView _ StandardSystemView new label: 'Counter'.
        topView minimumSize: 160 @ 60.
        topView addSubView: aCounterView
                in: (0 @ 0 extent: 1 @ 0.6)
                borderWidth: 2.
        switch _ Button newOff.         "increment button"
                switch onAction: [aCounterView model increment].
                aSwitchView _ SwitchView new model: switch.
                aSwitchView label: (Text string: 'increment' emphasis:
                                                                2) asDisplayText.
                aSwitchView insideColor: Form white.
        topView addSubView: aSwitchView
                in: (0 @ 0.6 extent: 0.5 @ 0.4)
                borderWidth: (2 @ 0 extent: 0 @ 2).
        switch _ Button newOff.         "decrement button"
                switch onAction: [aCounterView model decrement].
                aSwitchView _ SwitchView new model: switch.
                aSwitchView label: (Text string: 'decrement' emphasis:
                                                                2) asDisplayText.
                aSwitchView insideColor: Form white.
        topView addSubView: aSwitchView
```

```
                in: (0.5 @ 0.6 extent: 0.5 @ 0.4)
                borderWidth: (0 @ 0 extent: 2 @ 2).
        topView controller open! !

MouseMenuController subclass: #CounterController
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Demo-Counter'!

!CounterController methodsFor: 'initialize-release'!
initialize
     "Initialize a pop-up menu of commands for changing the value
                                                                of the model. "
      super initialize.
      self yellowButtonMenu: (PopUpMenu labelList: #((Increment Decrement ) ))
           yellowButtonMessages: #(increment decrement )! !

!CounterController methodsFor: 'control defaults'!
isControlActive
        "Take control when the blue button is not pressed."
        ^super isControlActive & sensor blueButtonPressed not! !

!CounterController methodsFor: 'menu messages'!
decrement
        "Subtract 1 from the value of the counter."
        self model decrement!
increment
        "Add 1 to the value of the counter."
        self model increment! !
Model subclass: #Counter
        instanceVariableNames: 'value '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Demo-Counter'!

!Counter methodsFor: 'initialize-release'!
initialize
        "Set the initial value to 0."
        self value: 0! !

!Counter methodsFor: 'accessing'!
getValue
        "Return the current value of 'value'"
        ^value!

setValue: aNumber
        "Set the value of 'value' to be aNumber"
        value := aNumber!

value
        "Answer the current value of the receiver."
        ^value!

value: aNumber
        "Set the counter to aNumber."
        value _ aNumber.
     self changed! !

!Counter methodsFor: 'operations'!
decrement
        "Subtract 1 from the value of the counter."
        self value: self value - 1!

increment
        "Add 1 to the value of the counter."
        self value: self value + 1!

printOn: aStream
   aStream nextPutAll: 'a CounterHolder with value ', self value printString! !

!Counter methodsFor: 'printing'! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

Counter class
        instanceVariableNames: ''!

!Counter class methodsFor: 'instance creation'!
new

"Answer an initialized instance of the receiver."
^super new initialize! !
```

End Listing

Listing One *(Text begins on page 62.)*

```
/*************************************************************************
 * The SEG4G Library by Al Williams. ASMFUNC.H--This header allows an array to
 * be executed as assembly code. The routine is called in far model regardless
 * of the model the C program is compiled in.
 *************************************************************************/

#define asmfunc *(int (far *)())
```

End Listing One

Listing Two

```
/*************************************************************************
 * SEG4G.H--Header for SEG4G Library  -- Williams                        *
 *************************************************************************/
typedef unsigned long LPTR;

/* set this variable to 0 for normal 386, 1 for Intel INBOARD 386/PC */
extern int inboard;

/* Function prototypes */
LPTR seg_to_linear(void far *p);
void far *linear_to_seg(LPTR lin);
void extend_seg(void);
void a20(int flag);
unsigned int big_read(LPTR address);
void big_write(LPTR address,unsigned int byte);
void big_xfer(LPTR src, LPTR dst, unsigned long count);

unsigned int ext_size(void);
LPTR ext_alloc(unsigned size);
LPTR ext_realloc(unsigned size);
int ext_free(int exitflag);
```

End Listing Two

Listing Three

```
/*************************************************************************
 * The SEG4G Library by Al Williams. SEG4G.C--These subroutines will allow *
 * a 386 to access a linear address space of 4 Gigabytes. You may select  *
 * one of three methods for incorporating assembly language subroutines    *
 * into the C programs. The three methods are:                             *
 *   ASM   - Use Microsoft's MASM 5.1                                      *
 *   DATA  - Use the asmfunc macro defined in ASMFUNC.H                    *
 *   POWER - Use the asm function present in POWER C                       *
 * You must select one of the three methods below:                         *
 *************************************************************************/
#define ASM   1
#define DATA  2
#define POWER 3
/* make your selection here: */
#define METHOD DATA

/* If using an INTEL INBOARD 386/PC set this variable to 1 */
int inboard=0;

#include <dos.h>
#include "seg4g.h"

/* Only include asmfunc.h if required */
#if METHOD==DATA
#include "asmfunc.h"
#endif

/* Keyboard controller defines */
#define RAMPORT   0x70
#define KB_PORT   0x64
#define PCNMIPORT 0xA0
#define INBA20    0x60
#define INBA20ON  0xDF
#define INBA20OFF 0xDD

/* Redefinitions for POWERC */
#ifdef __POWERC
#define _enable enable
#define _disable disable
#endif

/*************************************************************************
 * convert a far pointer to a linear address                             *
 *************************************************************************/
LPTR seg_to_linear(void far *p)
  {
  return (((unsigned long)FP_SEG(p))<<4)+FP_OFF(p);
  }

/*************************************************************************
 * convert a linear address to a far pointer                             *
 *************************************************************************/
void far *linear_to_seg(LPTR lin)
  {
  void far *p;
  FP_SEG(p)=(unsigned int)(lin>>4);
  FP_OFF(p)=(unsigned int)(lin&0xF);
  return p;
  }

/* Global descriptor table */
struct _GDT
   {
   unsigned int limit;
   unsigned int base;
   unsigned int access;
   unsigned int hi_limit;
   };

static struct _GDT  GDT[2] =
   {
     {0,0,0,0},                    /* unusable GDT slot 0 */
     {0xFFFF,0,0x9200,0x8F}        /* 4 Gig data segment */
   };

/* FWORD pointer to GDT */
struct fword
   {
   unsigned int limit;
   unsigned long linear_add;
   };

static struct fword  gdtptr;   /* fword ptr to gdt */

#if METHOD==POWER || METHOD==DATA

/* Protected mode assembly language routine */
static unsigned char code[]={
#if METHOD==DATA
        0x55,                 /* PUSH BP                */
        0x89, 0xe5,           /* MOV BP,SP              */
        0x1e,                 /* PUSH DS                */
        0xc5, 0x5e, 0x06,     /* LDS BX,[BP+6]          */
        0x0F, 0x01, 0x17,     /* LGDT FWORD PTR [BX]    */
        0x1f,                 /* POP DS                 */
        0x0f, 0x20, 0xc0,     /* MOV EAX,CR0            */
        0x0c, 0x01,           /* OR AL,1                */
        0x0f, 0x22, 0xc0,     /* MOV CR0, EAX           */
        0xeb, 0x00,           /* JMP SHORT 00           */
        0xbb, 0x08, 0x00,     /* MOV BX,8               */
        0x8e, 0xeb,           /* MOV GS,BX              */
        0x8e, 0xc3,           /* MOV ES,BX              */
        0x24, 0xfe,           /* AND AL,0FEH            */
        0x0f, 0x22, 0xc0,     /* MOV CR0,EAX            */
        0x5d,                 /* POP BP                 */
        0xcb};                /* RETF                   */
#else
        0x0f, 0x01, 0x17,     /* LGDT [BX]              */
        0x0f, 0x20, 0xc0,     /* MOV EAX,CR0            */
        0x0c, 0x01,           /* OR AL,1                */
        0x0f, 0x22, 0xc0,     /* MOV CR0,EAX            */
        0xEB, 0x00,           /* JMP SHORT 0            */
        0xbb, 0x08, 0x00,     /* MOV BX,8               */
        0x8e, 0xeb,           /* MOV GS,BX              */
        0X8e, 0xc3,           /* MOV ES,BX              */
        0x24, 0xfe,           /* AND AL,0FEH            */
        0x0f, 0x22, 0xc0,     /* MOV CR0,EAX            */
        0xC3 };               /* RETN                   */
#endif
#endif

/*************************************************************************
 * Adjust the ES and GS registers' limit to 4GB                          *
 *************************************************************************/
void extend_seg()
  {

  /* compute linear address and limit of GDT */
    gdtptr.linear_add=seg_to_linear((void far *)GDT);
    gdtptr.limit=15;

  /* disable regular interrupts */
    _disable();

  /* disable NMI */
    if (inboard)
      outp(PCNMIPORT,0);
    else
      outp(RAMPORT,inp(RAMPORT)|0x80);

  /* call protected mode code */
  #if METHOD==ASM
    protsetup(&gdtptr);
  #elif METHOD==DATA
    (asmfunc code)((void far *)&gdtptr);
  #else
    asm(code,&gdtptr);
  #endif
  /* Turn interrupts back on */
    _enable();

  /* Turn NMI back on */
    if (inboard)
      outp(PCNMIPORT,0x80);
    else
      outp(RAMPORT,inp(RAMPORT)&0x7F);
    }

  /* macro to clear keyboard port */
  #define keywait() { while (inp(KB_PORT)&2); }

/*************************************************************************
 * General purpose routine to allow A20 (flag=1) or disable A20 (flag=0) *
 *************************************************************************/
void a20(int flag)
  {
  if (inboard)
    {
    outp(INBA20,flag?INBA20ON:INBA20OFF);
    }
  else
    {
    keywait();
    outp(KB_PORT,flag?0xbc:0xb4);
    keywait();
    outp(KB_PORT,flag?0xbc:0xb4);
    keywait();
    }
  }
```

```
#if METHOD==DATA || METHOD==POWER
/* Assembly code to read a byte */
static unsigned char rcode[]={
#if METHOD==DATA
        0x55,                           /* PUSH BP              */
        0x89, 0xe5,                     /* MOV BP,SP            */
        0x33, 0xc0,                     /* XOR AX,AX            */
        0x8e, 0xe8,                     /* MOV GS,AX            */
        0x66, 0x8b, 0x46, 0x06,         /* MOV EAX,[BP+6]       */
        0x65, 0x67, 0x8a, 0x00,         /* MOV AL,GS:[EAX]      */
        0x32, 0xe4,                     /* XOR AH,AH            */
        0x5d,                           /* POP BP               */
        0xcb};                          /* RETF                 */
#else
        0x31, 0xC0,                     /* XOR AX,AX            */
        0x65, 0x8e, 0xC0,               /* MOV GS,AX            */
        0x66, 0x8b, 0x07,               /* MOV EAX,[BX]         */
        0x65, 0x67, 0x8a, 0x00,         /* MOV AL,GS:[EAX]      */
        0xC3 };                         /* RETN                 */
#endif

/* Assembly code to write a byte */
static unsigned char wcode[]={
#if METHOD==DATA
        0x55,                           /* PUSH BP              */
        0x89, 0xe5,                     /* MOV BP,SP            */
        0x33, 0xc0,                     /* XOR AX,AX            */
        0x8e, 0xe8,                     /* MOV GS,AX            */
        0x66, 0x8b, 0x46, 0x06,         /* MOV EAX,[BP+6]       */
        0x8b, 0x5e, 0x0a,               /* MOV BX,[BP+10]       */
        0x65, 0x67, 0x88, 0x18,         /* MOV GS:[EAX],BL      */
        0x5d,                           /* POP BP               */
        0xcb};                          /* RETF                 */
#else
        0x31, 0xC0,                     /* XOR AX,AX            */
        0x65, 0x8e, 0xC0,               /* MOV GS,AX            */
        0x66, 0x8b, 0x07,               /* MOV EAX,[BX]         */
        0x65, 0x67, 0xc6, 0x00, 0x00,   /* MOV GS:[EAX],??      */
        0xC3 };                         /* RETN                 */
#endif

/* Assembly code to block move bytes */
static unsigned char xcode[]={
#if METHOD==DATA
        0x55,                           /* PUSH BP              */
        0x89, 0xe5,                     /* MOV BP,SP            */
        0x06,                           /* PUSH ES              */
        0x56,                           /* PUSH SI              */
        0X57,                           /* PUSH DI              */
        0x33, 0xc0,                     /* XOR AX,AX            */
        0x8e, 0xC0,                     /* MOV ES,AX            */
        0X66, 0X8B, 0X76, 0X06,         /* MOV ESI,[BP+6]       */
        0X66, 0X8B, 0X7E, 0X0A,         /* MOV EDI,[BP+0A]      */
        0X66, 0X8B, 0X4E, 0X0E,         /* MOV ECX,[BP+0E]      */
        0XFC,                           /* CLD                  */
        0X67, 0XE3, 0X29,               /* JECX XEXIT           */
        0XF7, 0XC6, 0X03, 0X00,         /* TEST SI,3            */
        0x74, 0x0D,                     /* JZ XMAIN             */
        0XF7, 0XC7, 0X03, 0X00,         /* TEST DI,3            */
        0x74, 0x07,                     /* JZ XMAIN             */
        0X67, 0X26, 0XA4,               /* MOVSB ES:            */
        0x66, 0X49,                     /* DEC ECX              */
        0XEB, 0XEA,                     /* JMP XTEST            */
        0X51,                           /* PUSH CX              */
        0X66, 0XC1, 0XE9, 0X02,         /* SHR ECX,2            */
        0XF3, 0X67, 0X66, 0X26, 0XA5,   /* REP MOVSD ES:        */
        0X59,                           /* POP CX               */
        0X80, 0XE1, 0X03,               /* AND CX,3             */
        0XE3, 0X06,                     /* JCXZ XEXIT           */
        0X67, 0X26, 0XA4,               /* MOVSB ES:            */
        0X49,                           /* DEC CX               */
        0XEB, 0XF8,                     /* JMP XBYTE            */
        0X5F,                           /* POP DI               */
        0X5E,                           /* POP SI               */
        0x07,                           /* POP ES               */
        0X5D,                           /* POP BP               */
        0XCB};                          /* RETF                 */
#else
        0x55,                           /* PUSH BP              */
        0x89, 0xe5,                     /* MOV BP,SP            */
        0x06,                           /* PUSH ES              */
        0x33, 0xc0,                     /* XOR AX,AX            */
        0x8E, 0xC0,                     /* MOV ES,AX            */
        0X66, 0XBE,                     /* MOV ESI,             */
          0X00, 0X00, 0x00, 0x00,       /* SRC ADDRESS          */
        0x66, 0xBF,                     /* MOV EDI,             */
          0x00, 0x00, 0x00, 0x00,       /* DST ADDRESS          */
        0x66, 0xB9,                     /* MOV ECX,             */
          0x00, 0x00, 0x00, 0x00,       /* COUNT                */
        0XFC,                           /* CLD                  */
        0X67, 0XE3, 0X29,               /* JECX XEXIT           */
        0XF7, 0XC6, 0X03, 0X00,         /* TEST SI,3            */
        0x74, 0x0D,                     /* JZ XMAIN             */
        0XF7, 0XC7, 0X03, 0X00,         /* TEST DI,3            */
        0x74, 0x07,                     /* JZ XMAIN             */
        0X67, 0X26, 0XA4,               /* MOVSB ES:            */
        0x66, 0X49,                     /* DEC ECX              */
        0XEB, 0XEA,                     /* JMP XTEST            */
        0X51,                           /* PUSH CX              */
        0X66, 0XC1, 0XE9, 0X02,         /* SHR ECX,2            */
        0XF3, 0X67, 0X66, 0X26, 0XA5,   /* REP MOVSD ES:        */
        0X59,                           /* POP CX               */
        0X80, 0XE1, 0X03,               /* AND CX,3             */
        0XE3, 0X06,                     /* JCXZ XEXIT           */
        0X67, 0X26, 0XA4,               /* MOVSB ES:            */
        0X49,                           /* DEC CX               */
        0XEB, 0XF8,                     /* JMP XBYTE            */
        0x07,                           /* POP ES               */
        0X5D,                           /* POP BP               */
        0xC3,                           /* RETN                 */
#endif
```

```
/************************************************************************
 * Read a single byte from extended memory given a linear address       *
 ************************************************************************/
unsigned int big_read(LPTR address)
  {
#if METHOD==DATA
  return (asmfunc rcode)(address);
#else
  return asm(rcode,&address)&0xFF;
#endif
  }

/************************************************************************
 * Write a single byte to extended memory given a linear address        *
 ************************************************************************/
void big_write(LPTR address,unsigned int byte)
  {
#if METHOD==DATA
  (asmfunc wcode)(address,byte);
#else
  wcode[12]=byte;
  asm(wcode,&address);
#endif
  }

/************************************************************************
 * Block move a number of bytes from one area to another                *
 ************************************************************************/
void big_xfer(LPTR src,LPTR dst,unsigned long count)
  {
#if METHOD==DATA
  (asmfunc xcode)(src,dst,count);
#else
  *(LPTR *)&xcode[10]=src;
  *(LPTR *)&xcode[16]=dst;
  *(unsigned long *)&xcode[22]=count;
  asm(xcode,(void *)0);
#endif
  }

#endif
```

End Listing Three

Listing Four

```
.MODEL LARGE,C

.386P

.CODE

; SEG51.ASM
; Routine to goto protected mode and reset ES and GS registers to 4GB

IF @DataSize
protsetup proc fpointer:dword,c
          push ds
          lds bx,fpointer
ELSE
protsetup proc fpointer:word,c
          mov bx,fpointer
ENDIF
          lgdt fword ptr [bx]                   ; Load GDT
IF @DataSize
          pop ds
ENDIF
          mov eax,cr0                           ; Goto prot mode
          or al,1
          mov cr0,eax
          jmp short nxtlbl                      ; Purge instruction
nxtlbl:   mov bx,8                              ; prefetch
          mov gs,bx                             ; Load gs/es
          mov es,bx
          and al,0feh                           ; Go back to real mode
          mov cr0,eax
          ret
protsetup endp

; Read a byte from an LPTR
big_read proc address:dword,c
         xor ax,ax                              ; zero GS
         mov gs,ax
         mov eax,address                        ; Load LPTR
         mov al,gs:[eax]                        ; Load byte
         xor ah,ah                              ; Zero AH
         ret
big_read endp

; Write a byte to an LPTR address
big_write proc address:dword, byt:word,c
          xor ax,ax                             ; Zero GS
          mov gs,ax
          mov eax,address                       ; Load LPTR
          mov bx,byt                            ; Load byte
          mov byte ptr gs:[eax],bl              ; Store byte -> LPTR
          ret
big_write endp

; Block move bytes between LPTR's
big_xfer proc source:dword, dest:dword, count:dword,c
         push es
         push si
         push di
         xor ax,ax                              ; Zero ES
```

(Listing continued on page 112.)

Listing Four *(Listing continued, text begins on page 62.)*

```
        mov es,ax
        mov esi,source                          ; load source buffer
        mov edi,dest                            ; load dest buffer
        mov ecx,count                           ; load count
        cld
; The following code tries its best to make efficient moves
; by trying to move bytes until word alignment is achived
xtest:
        jecxz xexit                             ; done?
        test si,3                               ; SI word aligned?
        jz short xmain
        test di,3                               ; DI word aligned?
        jz short xmain
        movs es:[esi],byte ptr es:[edi]         ; Move a byte
        dec ecx                                 ; update count
        jmp short xtest                         ; Recheck alignments
xmain:
        push cx
        shr ecx,2                               ; Calculate number of dwords
                                                ; And move all of them
        rep movs dword ptr es:[esi],dword ptr es:[edi]
        pop cx
        and cl,3                                ; Move left over bytes
xbyte:  jcxz xexit                              ; If any
        movs es:[esi],byte ptr es:[edi]
        dec cx
        jmp short xbyte
xexit:
        pop di
        pop si
        pop es
        ret
big_xfer endp

         end
```

End Listing Four

Listing Five

```
/*************************************************************************
 * The SEG4G Library by Al Williams. EXTMEM.C--These subroutines manage  *
 * top down allocation of extended memory.                               *
 *************************************************************************/
#include <dos.h>
#include "seg4g.h"

static void far *old15;
static int installed=0;
static unsigned e_size, e_alloc;

/* redefinitions for POWERC */
#ifdef __POWERC
#define _FAR
#define _dos_getvect(n) getvect(n)
#define _dos_setvect(n,p) setvect(n,p)

/* This is a kludge to get POWERC to chain to the next level of interrupt */
unsigned __chain[14]= { 0x559c,0xe589,0xb850,0, 0x4687, 0x87fe, 0x46,\
                        0xb850, 0, 0x4687, 0x5d00, 0xeafa, 0, 0 };
void far *__cptr;

#define _chain_intr(ptr) { __cptr=(void far *)__chain; \
                           __chain[12]=FP_OFF(ptr); \
                           __chain[13]=FP_SEG(ptr); __chain[3]=Rip;\
                           __chain[8]=Rcs; Rcs=FP_SEG(__cptr);\
                           Rip=FP_OFF(__cptr);\
                           return; }

#define INTREGS unsigned Rbp, unsigned Rdi, \
                unsigned Rsi, unsigned Rds, \
                unsigned Res, unsigned Rdx, \
                unsigned Rcx, unsigned Rbx, \
                unsigned Rax, unsigned Rip, \
                unsigned Rcs, unsigned Rflags
#else
#define _FAR far
#define INTREGS unsigned Res, unsigned Rds, \
                unsigned Rdi, unsigned Rsi, \
                unsigned Rbp, unsigned Rsp, \
                unsigned Rbx, unsigned Rdx, \
                unsigned Rcx, unsigned Rax
#endif

/* private routine to capture requests for extended memory size */
static void interrupt _FAR trap15(INTREGS)
  {
  if ((Rax&0xFF00) != 0x8800)
    _chain_intr(old15);
  Rax=e_size;
  return;
  }

/*************************************************************************
 * Get extended memory size (in K) from BIOS                             *
 *************************************************************************/
unsigned int ext_size()
  {
  union REGS r;
  r.h.ah=0x88;
  int86(0x15,&r,&r);
  return r.x.ax;
  }

/*************************************************************************
 * Allocate memory in 1K blocks, returns start address of block or       *
 * (LPTR) -1 if unable to allocate memory                                *
 *************************************************************************/
LPTR ext_alloc(unsigned size)
  {
  if (installed)
    return ext_realloc(size+e_alloc);
  e_alloc=size;
  e_size=ext_size();
  if (e_size<size) return (LPTR) -1L;
  e_size-=size;
  old15=_dos_getvect(0x15);
  _dos_setvect(0x15,trap15);
  installed=1;
  return 0x100000+e_size*1024;
  }

/*************************************************************************
 * Attempt to change the size of an allocated block (size in K).         *
 * Returns start address or (LPTR) -1 if unsuccessful                    *
 *************************************************************************/
LPTR ext_realloc(unsigned size)
  {
  if (!installed)
    return ext_alloc(size);
  if (size>e_alloc+e_size) return (LPTR)-1L;
  if (size<e_alloc)
    {
    e_size+=e_alloc-size;
    e_alloc=size;
    }
  else if (size>e_alloc)
    {
    if (_dos_getvect(0x15)!=trap15)
      return (LPTR) -1L;
    e_size-=size-e_alloc;
    e_alloc=size;
    }
  return 0x100000+e_size*1024;
  }

/*************************************************************************
 * Free the extended block. Always call before exiting your program!     *
 * If exitflag is set, the INT 15 trap will be reset. If another program *
 * has captured INT 15, ext_free will return a -1. If you call with      *
 * exitflag == 0, and another program has captured INT 15, the vector is *
 * not reset and ext_free returns a 1. Otherwise, ext_free returns 0 and *
 * releases INT 15                                                       *
 *************************************************************************/
int ext_free(int exitflag)
  {
  int rc=0;
  if (!installed) return rc;
  if (_dos_getvect(0x15)==trap15||exitflag)
    {
    if (_dos_getvect(0x15)!=trap15) rc=-1;
    installed=0;
    _dos_setvect(0x15,old15);
    }
  else
    {
    e_size+=e_alloc;
    e_alloc=0;
    rc=1;
    }
  return rc;
  }
```

End Listing Five

Listing Six

```
/*************************************************************************
 * TEST.C--Example program for the SEG4G library                         *
 *************************************************************************/
#include <stdio.h>
#include <ctype.h>
#include <signal.h>
#include "seg4g.h"
main()
  {
  LPTR ad,aptr;
  int ct=1024,i;
  int data=0xAA;
/* Ignore breaks */
  signal(SIGINT,SIG_IGN);
  printf("%dK of extended memory available\n",ext_size());
/* allocate 1K of extended */
  ad=ext_alloc(1);
  if (ad==-1L)
    {
    printf("Not enough extended memory. Only %dK available.\n",ext_size());
    exit(1);
    }
  printf("1K of extended mem allocated at %81X. %dK remains.\n",ad,ext_size());
/* Make 4GB segments */
  extend_seg();
/* Turn on A20 */
  a20(1);
/* Write data to block */
  aptr=ad;
  for (i=0;i<ct;i++)
    {
    big_write(aptr++,data);
    }
  printf("Data written to extended memory\n\n");
/* Read it back */
```

(Listing continued on page 115.)

Listing Six *(Listing continued, text begins on page 62.)*

```
  aptr=ad;
  for (i=0;i<ct;i++)
    {
    if (big_read(aptr++)!=data)
      {
      printf("Error reading extended memory\n\n");
      ext_free(1);
      a20(0);
      exit(1);
      }
    }
  printf("Data read back OK.\nExpanding allocation to 2K\n");
/* Expand memory allocation for no good reason */
  ad=ext_realloc(2);
  if (ad==-1L)
    {
    printf("Not enough extended memory. Only %dK is available.\n",ext_size());
    exit(1);
    }
  printf("2K of extended mem allocated at %81X. %dK remains.\n",ad,ext_size());
/* Free memory */
  ext_free(1);
  printf("Extended memory freed. %dK Available.\n",ext_size());
/* Enter memory examine loop */
  while (1)
    {
    printf("Enter ^Z to quit.\nAddress and count? ");
    if (scanf("%li %i",&ad,&ct)!=2)
      {
      a20(0);
      exit(0);
      }
    while (ct--)
      {
      data=big_read(ad++);
      printf("%c",isgraph(data)?data:'.');
      }
    printf("\n\n");
    }
  }
```

End Listing Six

Listing Seven

```
/*************************************************************************
 * BLKTEST.C--Example block move program for the SEG4G library           *
 *************************************************************************/
#include <stdio.h>
#include <dos.h>
#include "seg4g.h"

/* Set COLOR to 0 if you have a monochrome monitor */
#define COLOR 1

#define SCREEN_SIZE 4000
#define ALIGN_SIZE 3

unsigned char pattern[SCREEN_SIZE+ALIGN_SIZE];

main()
  {
  LPTR data,screen;
  unsigned char far *p;
  int i;
  extend_seg();
#if COLOR
  screen=0xb8000;
#else
  screen=0xb0000;
#endif
  p=pattern;
/* align to nearest 4 byte boundry */
/* This isn't required, but does make big_xfer() more efficient */
 while (FP_OFF(p)&3) p++;
  data=seg_to_linear(p);
  for (i=0;i<SCREEN_SIZE;i+=4)
    {
    p[i]='A';
    p[i+3]=p[i+1]=0x70;
    p[i+2]='B';
    }
  big_xfer(data,screen,(unsigned long)SCREEN_SIZE);
  }
```

End Listings

Listing One *(Text begins on page 72.)*

```
DEFINT A-Z
DIM t!(28)
OPEN "basoldpb.tim" FOR OUTPUT AS #1
'time for a raw integer loop, executed 1,000,000 times.
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER

FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  NEXT j
NEXT i
t!(0) = TIMER - t!

'time for a integer assignment loop, executed 1,000,000 times.
y = 5: z = -5
PRINT "Int =";
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x = y
  x = z
  NEXT j
NEXT i
t!(1) = (TIMER - t! - t!(0)) / 2

'time for 1,000,000 integer adds
y = 5: z = -5
PRINT "+";
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x = x + y
  x = x + z
  NEXT j
NEXT i
t!(2) = (TIMER - t! - t!(0)) / 2 - t!(1)

'time for 1,000,000 integer subtracts
PRINT "-";
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x = x - y
  x = x - z
  NEXT j
NEXT i
t!(3) = (TIMER - t! - t!(0)) / 2 - t!(1)

'time for 1,000,000 integer multiplies
k = 7
PRINT "*";
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x = k * j
  NEXT j
NEXT i
t!(4) = TIMER - t! - t!(1)

'time for 1,000,000 integer divides
PRINT "\"
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x = i \ j
  NEXT j
NEXT i
t!(5) = TIMER - t! - t!(1)

'time for 100,000 string assignments
PRINT "String=";
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 100
  x$ = "abcdefghijklmnopqrstuvwxyz"
  x$ = "zyxwvutsrqponmlkjihgfedcba"
  NEXT j
NEXT i
t!(6) = (TIMER - t! - t!(0) / 10) / 2

'time for 100,000 string MID$ operations
x$ = "abcdefghijklmnopqrstuvwxyz"
k = 17
PRINT "Mid$";
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 100
  MID$(x$, k, 1) = "d"
  NEXT j
NEXT i
t!(7) = TIMER - t! - t!(0) / 10

'time for 10,000 string concatenations
x$ = ""
PRINT "+"
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 10000
   x$ = x$ + "a"
NEXT i
t!(8) = TIMER - t! - t!(6) / 10 - t!(0) / 100

'time for a single-precision assignment loop, executed 1,000,000 times.
y! = 5!: z! = -5!
PRINT "Single=";
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x! = y!
  x! = z!
  NEXT j
NEXT i
t!(9) = (TIMER - t! - t!(0)) / 2

'time for 1,000,000 single-precision adds
PRINT "+";
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
```

```
  FOR j = 1 TO 1000
    x! = x! + y!
    x! = x! + z!
  NEXT j
NEXT i
t!(10) = (TIMER - t! - t!(0)) / 2 - t!(9)

'time for 1,000,000 single-precision subtracts
x! = 0!
PRINT "-";
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
    x! = x! - y!
    x! = x! - z!
  NEXT j
NEXT i
t!(11) = (TIMER - t! - t!(0)) / 2 - t!(9)

'time for 100,000 single-precision multiplies
x! = 1!
PRINT "*";
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 100
    x! = x! * 1.00001
  NEXT j
NEXT i
t!(12) = TIMER - t! - t!(0) / 10 - t!(9) / 10

'time for 100,000 single-precision divides
PRINT "/";
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 100
    x! = x! / 1.00001
  NEXT j
NEXT i
t!(13) = TIMER - t! - t!(0) / 10 - t!(9) / 10

'error in single-precision multiply/divide
t!(14) = x! - 1!

'time for 10,000 single-precision exponentiations
x! = 100!
PRINT "^"
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 10000
  x! = x! ^ .999999
NEXT i
t!(15) = TIMER - t! - t!(0) / 100 - t!(9) / 100
'time for a double-precision assignment loop, executed 1,000,000 times.
y# = 5.5#: z# = -5.5#
PRINT "Double=";
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x# = y#
  x# = z#
  NEXT j
NEXT i
t!(16) = (TIMER - t! - t!(0)) / 2

'time for 1,000,000 double-precision adds
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
PRINT "+";
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
    x# = x# + y#
  NEXT j
NEXT i
t!(17) = TIMER - t! - t!(16) - t!(0)

'time for 1,000,000 double-precision subtracts
x# = 0#
PRINT "-";
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
    x# = x# - y#
  NEXT j
NEXT i
t!(18) = TIMER - t! - t!(16) - t!(0)

'time for 100,000 double-precision multiplies
x# = 1#
PRINT "*";
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 100
    x# = x# * 1.00001#
  NEXT j
NEXT i
t!(19) = (TIMER - t! - t!(0) / 10) - t!(16) / 10

'time for 100,000 double-precision divides
PRINT "/";
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 100
    x# = x# / 1.00001#
  NEXT j
NEXT i
t!(20) = (TIMER - t! - t!(0) / 10) - t!(16) / 10

'error in double-precision multiply/divide
t!(21) = x# - 1#
```

(Listings continued on page 118.)

Listing One *(Listing continued, text begins on page 72.)*

```
'time for 10,000 double-precision exponentiations
x# = 100#
PRINT "^"
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 10000
  x# = x# ^ .999999#
NEXT i
t!(22) = (TIMER - t! - t!(0) / 100) - t!(16) / 100


'following are logical comparisons and operators

'time for 1,000,000 integer comparisons
x = 0
PRINT "Int compar"
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
    IF i < x THEN x = 1
  NEXT j
NEXT i
t!(23) = TIMER - t! - t!(0)

'time for 1,000,000 single-precision comparisons
x! = 5!: y! = 3.333
PRINT "Single compar"
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
    IF x! < y! THEN x = 1
  NEXT j
NEXT i
t!(24) = TIMER - t! - t!(0)

'time for 1,000,000 double-precision comparisons
x# = 5#: y# = 3.333#
PRINT "Double compar"
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
    IF x# < y# THEN x = 1
  NEXT j
NEXT i
t!(25) = TIMER - t! - t!(0)

'is there short-circuit expression evaluation?
'integer loop, 1,000,000 times
PRINT "Short"
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
    IF i < 0 AND j < 10 THEN x = 1
  NEXT j
NEXT i
t!(26) = TIMER - t! - t!(0)
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
    IF j < 10 AND i < 0 THEN x = 1
  NEXT j
NEXT i
t!(27) = TIMER - t! - t!(0)
'Note: if the two times are appreciably different, some optimization has been
'done.  The first time should be shorter than the second.

'screen output: print 1,000 70-byte strings
x$ = STRING$(70, 66)
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  PRINT x$
NEXT i
t!(28) = TIMER - t! - t!(0) / 1000


'print results of benchmark
PRINT #1, "Empty integer loop, 1,000,000 iterations:"; TAB(46); t!(0)
PRINT #1, "1,000,000 integer assignments:"; TAB(46); t!(1)
PRINT #1, "1,000,000 integer additions:"; TAB(46); t!(2)
PRINT #1, "1,000,000 integer subtractions:"; TAB(46); t!(3)
PRINT #1, "1,000,000 integer multiplications:"; TAB(46); t!(4)
PRINT #1, "1,000,000 integer divisions:"; TAB(46); t!(5)
PRINT #1, "100,000 string assignments:"; TAB(46); t!(6)
PRINT #1, "100,000 string MID$ operations:"; TAB(46); t!(7)
PRINT #1, "10,000 string concatenations:"; TAB(46); t!(8)
PRINT #1, "1,000,000 single-precision assignments:"; TAB(46); t!(9)
PRINT #1, "1,000,000 single-precision additions:"; TAB(46); t!(10)
PRINT #1, "1,000,000 single-precision subtractions:"; TAB(46); t!(11)
PRINT #1, "100,000 single-precision multiplications:"; TAB(46); t!(12)
PRINT #1, "100,000 single-precision divisions:"; TAB(46); t!(13)
PRINT #1, "Error in 100,000 single-precision mult/div:"; TAB(46); t!(14)
PRINT #1, "10,000 single-precision exponentiations:"; TAB(46); t!(15)
PRINT #1, "1,000,000 double-precision assignments:"; TAB(46); t!(16)
PRINT #1, "1,000,000 double-precision additions:"; TAB(46); t!(17)
PRINT #1, "1,000,000 double-precision subtractions:"; TAB(46); t!(18)
PRINT #1, "100,000 double-precision multiplications:"; TAB(46); t!(19)
PRINT #1, "100,000 double-precision divisions:"; TAB(46); t!(20)
PRINT #1, "Error in 100,000 double-precision mult/div:"; TAB(46); t!(21)
PRINT #1, "10,000 double-precision exponentiations:"; TAB(46); t!(22)
PRINT #1, "1,000,000 integer comparisons:"; TAB(46); t!(23)
PRINT #1, "1,000,000 single-precision comparisons:"; TAB(46); t!(24)
PRINT #1, "1,000,000 double-precision comparisons:"; TAB(46); t!(25)
PRINT #1, "1,000,000 conditional integer assignments:"; TAB(46); t!(26)
PRINT #1, "1,000,000 conditional assignments (reversed):"; TAB(46); t!(27)
PRINT #1, "Print 1,000 70-byte strings to the screen:"; TAB(46); t!(28)
END
```

End Listing One

Listing Two *(Text begins on page 72.)*

```
DEFLNG A-Z
DIM t!(28)
OPEN "basnewpb.tim" FOR OUTPUT AS #1
MAP x1$$ * 1
MAP x26$$ * 26
MAP x70$$ * 70
MAP x10000$$ * 10000

'time for a raw long integer loop, executed 1,000,000 times.
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  NEXT j
NEXT i
t!(0) = TIMER - t!

'time for 1,000,000 long integer assignments.
y = 5&: z = -5&: PRINT "Long int="
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x = y
  x = z
  NEXT j
NEXT i
t!(1) = (TIMER - t! - t!(0)) / 2

'time for 1,000,000 long integer adds
PRINT "+"
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x = x + y
  NEXT j
NEXT i
t!(2) = TIMER - t! - t!(1)

'time for 1,000,000 long integer subtracts
x = 0
PRINT "-"
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x = x - y
  NEXT j
NEXT i
t!(3) = TIMER - t! - t!(1)

'time for 1,000,000 long integer multiplies
PRINT "*"
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x = i * j
  NEXT j
NEXT i
t!(4) = TIMER - t! - t!(1)

'time for 1,000,000 long integer divides
PRINT "/"
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
  x = i \ j
  NEXT j
NEXT i
t!(5) = TIMER - t! - t!(1)

'time for 100,000 fixed string assignments
PRINT "Fixed string="
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 100
  x26$$ = "abcdefghijklmnopqrstuvwxyz"
  x26$$ = "zyxwvutrsqponmlkjihgfedcba"
  NEXT j
NEXT i
t!(6) = (TIMER - t! - t!(0) / 10) / 2

'time for 100,000 fixed string MID$ operations
k = 17
PRINT "Mid$"
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 100
        MID$(x26$$, k, 1) = "d"
  NEXT j
NEXT i
t!(7) = TIMER - t! - t!(0) / 10

'time for 10,000 fixed string "concatenations"
x$ = ""
PRINT "+"
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 10000
  MID$(x10000$$, i, 1) = "a"
NEXT i
t!(8) = TIMER - t! - t!(0) / 100

'following are logical comparisons and operators

'time for 1,000,000 long integer comparisons
x = 5&: y = -5&
PRINT "Long int compar"
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  FOR j = 1 TO 1000
      IF i < y THEN x = 1
```

(Listing continued on page 120.)

Listing Two *(Listing continued, text begins on page 72.)*

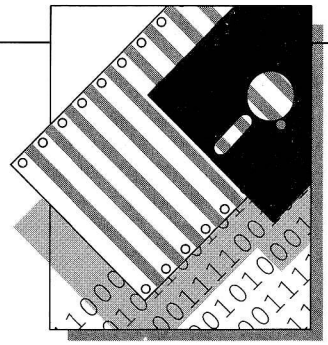
```
  NEXT j
NEXT i
t!(23) = TIMER - t! - t!(0)

'screen output: print 1,000 70-byte fixed strings
x70$$ = STRING$(70, 66)
t! = TIMER: WHILE t! = TIMER: WEND: t! = TIMER
FOR i = 1 TO 1000
  PRINT x70$$
NEXT i
t!(28) = TIMER - t! - t!(0) / 1000


'print results of benchmark
PRINT #1, "Raw long integer loop, 1,000,000 iterations:"; TAB(45); t!(0)
PRINT #1, "1,000,000 long integer assignments:"; TAB(45); t!(1)
PRINT #1, "1,000,000 long integer additions:"; TAB(45); t!(2)
PRINT #1, "1,000,000 long integer subtractions:"; TAB(45); t!(3)
PRINT #1, "1,000,000 long integer multiplications:"; TAB(45); t!(4)
PRINT #1, "1,000,000 long integer divisions:"; TAB(45); t!(5)
PRINT #1, "100,000 fixed string assignments:"; TAB(45); t!(6)
PRINT #1, "100,000 fixed string MID$ operations:"; TAB(45); t!(7)
PRINT #1, "10,000 fixed string concatenations:"; TAB(45); t!(8)
PRINT #1, "1,000,000 long integer comparisons:"; TAB(45); t!(23)
PRINT #1, "Print 1,000 70-byte strings to the screen:"; TAB(45); t!(28)
END
```

End Listing Two

Listing Three

```
defint a-z
cls
recordlength=152
print"Sample in-memory mailing list program demo for Dr. Dobbs Magazine."
open"r",1,"sample.dat",recordlength
field #1,recordlength as w$

'set up a flex string variable to hold the data fields
map rec$$*recordlength,10 as dte$$,25 as last$$,1 as middle$$,_
        20 as first$$,25 as address1$$,25 as address2$$,25 as city$$,_
        2 as state$$,9 as zip$$,10 as phone$$

total=lof(1)\recordlength               'number of records in the sample file
y=(fre(-1)-32000)\recordlength     'number of records that will fit in memory
if total>y then print"Not enough room for this file.":close:end

dim records$(1:y)               'ordinary string array will hold the records.
print"Please wait:reading the data file.";
propellor$="|/-\":where=pos(9)
for i=1 to total
        locate csrlin,where:print mid$(propellor$,1+i mod 4,1);
        get 1,i
        records$(i)=w$
next i
close 1
current=1

while 1
        cls
        print total;" total records of a maximum of";y
        print"Current record is #";current
        locate 4,1
        rec$$=records$(current)
        print "Date entered: ";dte$$;"     Phone:";phone$$
        print "Name: ";rtrim$(first$$);" ";middle$$;" ";last$$
        print "Address: ";address1$$
        print "Address: ";address2$$
        print "City, state, zip: ";rtrim$(city$$);" ";state$$;" ";zip$$

        locate 12,1
        print"E=enter new record D=delete current record Q=quit without saving"
        print"S=sort records       J=jump to record          X=save and exit"

        while not instat:wend:cmd$=inkey$
        locate 15,1:cmd$=ucase$(cmd$)

        select case cmd$
                case "E"
                        gosub entry
                case "D"
                        gosub deleterec
                case "Q"
                        gosub quit
                case "S"
                        gosub sortrecs
                case "J"
                        gosub jump
                case "X"
                        gosub exitprog
        end select
wend

entry:
        'insert new record at current position
        dte$$=date$
        line input"First name: ";first$$
        line input"Middle initial: ";middle$$
        line input"Last name: ";last$$
        line input"Address #1: ";address1$$
        line input"Address #2: ";address2$$
        line input"City: ";city$$
        line input"State: ";state$$
        line input"Zip: ";zip$$
        w$=rec$$
        array insert records$(current),w$
        total=total+1
        return

deleterec:
        'get rid of current record
        array delete records$(current)
        total=total-1:if total<current then current=total
        if current<1 then current=1
        return

quit:
        end

sortrecs:
        'sort by last name.
        array sort records$(1) for total, from 11 to 35
        return

jump:
        input"Jump to record: ";current
        if current>total then current=total
        if current<1 then current=1
        return

exitprog:
        print"Saving data file. ";
        open"o",1,"sample.dat"
        close 1
        open"r",1,"sample.dat",recordlength
        field #1,recordlength as w$
        where=pos(9)
        for i=1 to total
                lset w$=records$(i)
                put 1,i
                locate csrlin,where:print mid$(propellor$,1+i mod 4,1);
        next i
        close:print
        end
```

End Listings



Cooperation and Competition

This month I report on my visit to the April MacWorld Expo; raise some questions about Glasnost programming, ruminate on recent issues in chaos theory, fractals, and neural networks, and just barely justify the title of this month's column.

On the Paradigms Beat at MacWorld '90

The MacWorld show has now reached the same state that Comdex has been in for five years. Practically all the applications and development tools you could reasonably ask for are there, so the only question is, are they any good? That's not a question you can easily answer on the exhibit floor, so you collect the press kits, go to the parties, and schmooze. The show gave eve. evidence of a healthy market, ample opportunities for schmoozing, but few surprises.

Well, maybe object-oriented Prolog is a surprise. Quintus Computer Systems, of Mountain View, California, announced Version 3.0 of its Prolog compiler for the Mac, along with an object-oriented Prolog package. Prolog++/MacObject has objects, inheritance, methods, attributes, and daemons.

Michael Swaine

Apple announced Developer Tools Express, a mail-order service that allows developers to purchase release versions of Apple development tools without paying the APDA membership. To find out more about it, call or write Apple Developer Channels, Apple Computer Inc., 20525 Mariani Avenue, MS 33G, Cupertino, CA 95014-6299; 1-800-282-2734 (toll-free US); 1-800-637-0029 (toll-free Canada); 1- 408-562-3910 (international).

Apple also announced Version 2.0 of MacApp, the object-oriented application framework. It should be more robust than Version 1.0, which was really Version 2.0 of the product — the original being the Lisa Toolkit. At the Addison-Wesley booth, David Wilson, Larry Rosenstein, and Dan Shafer were signing copies of their *Programming with MacApp*, in the *Macintosh Inside Out* series edited by Scott Knaster. The book is based on Dave Wilson's courses on MacApp and object-oriented programming, and it looks good. You will need MacApp 2.0 to use the tutorial disk included with the book.

Roaming the aisles, I ran into a familiar Mac programmer who quoted new Apple COO Michael Spindler that there will be no more prima donnas at Apple. I said it showed that Spindler knew his limits in the charisma department, vis-à-vis Jean-Louis Gassé, and my programmer friend said no, it showed that Spindler didn't know anything about programmers.

A charisma gap at Apple could be a problem for the company. The Moscow model suggests that if you want your dictatorship to be perceived as benevolent, a little charisma doesn't hurt.

The 2-1/2-th Party Developer

Given that 1. Apple makes it clear to its third-party developers that deviations from the party line in matters such as the user interface are unwise, and 2. it expects the developers to see how this control benefits the developers themselves, it's not unfair to say that Apple wants its dictatorship to be perceived as benevolent. Apple isn't following any Moscow model, but it is pushing for a kind of Glasnost programming, a new openness that is supposed to result in cooperation within a competitive free market.

This new openness is the paradigm of cooperative programming, in which programs written by different companies work together harmoniously and communicate smoothly while the companies that produce them fight to drive each other out of business. There is some irony in such a plan coming along at the same time Apple is apparently fostering competition within the company by creating two distribution channels for developer tools, but no doubt APDA and Developer Tools Express will coexist amicably. Cooperative programming is not, of course, the exclusive province of the Mac's niche. (Apple may object that the Mac is not a niche product. Macht nichts.) But a benevolent dictatorship with control over the hardware platform and the operating system is in a better position than IBM or Microsoft to make it happen.

The basis for this new paradigm for Macintosh software development is Version 7 of the Mac operating system, due out later this year. Version 7 will support several levels of inter-application communication, from the existing copy and paste buffer to live copy and paste, network support, and store-and-forward communication. Apple argues that the rich set of interapplication communication tools in System 7 will lead to a new kind of software; to smaller, targeted tools that do one thing well, rather than massive integrated applications that try to do everything.

Apple believes that this will happen, is convinced that it had better happen, and is doing a lot to make it happen.

Claris has been talking up the new approach, saying in guest editorials in computer magazines that Macintosh software developers need to learn a new way of operating — a new cooperative spirit; the word Glasnost may not appear explicitly, but it's there between the lines. All this sounds better, one might argue, coming from a third-party developer rather than just from Apple. That is, if Apple software spinoff Claris can be called a third-party developer. The term "stalking horse" trots to mind.

Is Apple/Claris right? Is cooperative programming going to work? And if so, what will this cooperative programming be like?

Getting the Message

Here's what Apple's interapplication support under System 7 will include:

- The current copy and paste facility, restricted to data only, supporting various formats. The Scrap Manager will continue to provide this basic form of interapplication communication.
- Live copy and paste, in which changes made to the original (after copying and pasting) are reflected in all the copies. This is a new facility, provided by the Edition Manager in System 7.
- Message passing. Changes to the Event Manager will allow messages to be passed between applications rather than just within.
- AppleEvents. This is an Apple-defined set of standard messages that all applications will be expected to handle and that all are permitted to send. Message passing between applications will not be limited to AppleEvents, but AppleEvents will provide, in effect, a language for interapplication communication. It will also serve as the enabling technology for a future user scripting language. The user scripting language will allow users to control applications and the system itself with unprecedented (for the Macintosh) flexibility and power, and with a single, universal user interface.
- Lower-level tools for interapplication communication, including separate toolboxes for immediate and store-and-forward communication. Store-and-forward will allow the user or an application to leave a message for another application to handle even if the other application is currently in use or not running or temporarily in an inaccessible corner of a network.

Who invented the Mandelbrot set? There is a surly controversy brewing over credit in the field

The low-level interapplication communication facilities may be most useful between applications from a single vendor, but it seems likely that only AppleEvents, with its high level of control from Apple, will have the power to make cooperative programming between vendors work.

AppleEvents may also present a new kind of pressure to adapt. It could be the best tool Apple has for getting third-party developers to add interapplication communication features to their software fast. Here's a (perhaps) fanciful account of how that might work:

If another application sends your's an AppleEvents message for which you haven't written a handler, what hap-

(continued on page 127)

(continued from page 124)
pens is up to the system or to the messaging application, but not to you. In effect, part of the (extended) interface to your application has been taken out of your hands. An unanswered message to your spreadsheet application triggered from within a competitor's word processing application could conceivably result in a message to the user such as "Sorry. That spreadsheet program is too dumb to handle this simple request. Maybe you should consider buying our spreadsheet program, which always works smoothly with our word processor. And if you send in your copy of that worthless spreadsheet program, we'll give you a 20 percent discount."

One mathematician quoted lays all the blame for the controversy at IBM mathematician Benoit Mandelbrot's feet

Maybe it won't work that way, but it is true that the mere existence of interapplication communication will create a new window into every existing application. And it is true that the control over this window, this extension to the application's interface, will be in the hands of the developer of the application only if the application supports AppleEvents.

In any case, if your application is going to fail to handle these messages, it's much nicer if it can fail gracefully to handle them. This should be fairly easy. The set of AppleEvents messages will be well defined, and handling an AppleEvents message doesn't mean that you really have to do something with it. You can simple acknowledge it and do nothing, or send back some message of your own. That's code that could be written overnight.

Here comes the real pressure. What was formerly a large task, upgrading your application to take full advantage of a major new release of the operating system, has now become a collection of small features to add. The insidious aspect of this is that, once you've got a (dummy) handler for the message in your code, it nags you to make it real, as other people in your company are going to nag you to do the same.

Then there's user scripting. Apple has pushed back its attack on user scripting to Version 8.0, but others are not waiting for 8.0: Manufacturers of applications that already have user scripting will explore ways to incorporate AppleEvents into their scripting languages. Shortly, users are going to start sending messages to your application.

So yes, it looks like Apple's interapplication communication is going to catch on quickly, and it looks like cooperative programming will become a fact of life. And that is going to raise some new problems, not all of which are technical.

Back to the Future

Here are some follow-up observations on some of the more futuristic subjects I've discussed here recently: chaos theory, neural networks, and fractals.

Recently, I wrote here about chaos theory in human physiology, and suggested that the combination of neural nets and chaos theory could produce a powerful tool for exploring the behavior of certain types of complex systems. At the EURASIP Workshop on Neural Networks in Portugal this year, Steve Renals of Edinburgh University

(continued from page 127)
presented some results of work on chaos theory and neural networks. One of his conclusions was that chaos could fulfill an important role in associative memory systems by providing a "don't know" state. The conference papers are collected in "Neural Networks: EURASIP Workshop 1990 Proceedings," Springer-Verlag, 1990.

The April issue of *Scientific American* contained a piece on the use of neural networks in determining the structure of complex protein molecules. It's apparently no longer particularly difficult to map out the sequence of amino acids that make up a protein molecule, but the three-dimensional twists and turns of the chain-like molecule are very hard to identify. The twisting structure, which determines the properties of the protein, depends on the electrical forces between individual amino acids in the chain. A brute-force approach to determining the structure would simply examine all forces between all pairs of amino acids; and with very short chains, the brute-force approach works.

For most interesting proteins, though, the brute-force approach is useless. Terrence Sejnowski and Ning Qian of Johns Hopkins University have tackled the problem with a neural-network approach first used by Sejnowski and Charles Rosenberg in NETtalk, a neural net that learns to pronounce English words. NETtalk uses context to determine the pronunciation of individual phonemes (units of sound very roughly comparable to letters in written English). NETtalk has had some impressive success, and Sejnowski and Qian thought that the electrical interactions among segments of a protein could be modeled very much like context effects on phonemic units. The results, though limited, look promising, and Sejnowski thinks that neural nets have a real future in molecular biology.

The April issue of *Scientific American* also asks the question, Who invented the Mandelbrot set? There is a surly controversy brewing over credit in the field of fractals and particularly for this set, which has been called "the most complex object in mathematics." You compute the Mandelbrot set from the formula $z^2 + c$, where z and c are complex numbers, by assigning a constant value to c, setting z initially to 0, and feeding the output of the formula back into z iteratively. Pictures of the Rorschach-like Mandelbrot set have appeared here and elsewhere, and are distinctive, if only arguably attractive. Whatever its aesthetic merits, the Mandelbrot set's discovery is an important event in the history of mathematics, as the set has many applications, especially in the testing of complex dynamical systems. The study of such systems is the aforementioned chaos theory.

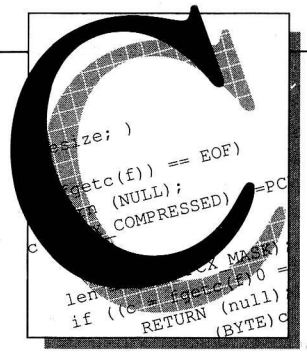
One mathematician quoted in the *Scientific American* piece lays all the blame for the controversy at IBM mathematician Benoit Mandelbrot's feet. Apparently all of the following people deserve some credit in the discovery and exploration of the Mandelbrot set: Pierre Fatou, who defined the set mathematically; Gaston Julia, Mandelbrot's teacher, who defined the Julia set, of which the Mandelbrot set is a generalization; John H. Hubbard and Adrien Douady, who have done much work on the set, Hubbard having produced Mandelbrot pictures three years before Mandelbrot; Robert Brooks and J. Peter Matelski, who published both an explicit mathematical formulation and a computer printout showing the familiar image of the set before Mandelbrot; and F. Riesz, who did related work about 40 years ago. No one challenges the importance of Mandelbrot's own contribution, but many are put off by Mandelbrot's aggressive self-promotion as the sole discoverer of the set. Former *DDJ* editor Randy Sutherland and I joined the put-off when we dealt with Mandelbrot while trying to get permission to reproduce a picture of the Mandelbrot set in *DDJ* a few years ago. As Douady said, "He loves to quote himself and he is very reluctant to quote others who aren't dead."

John Horgan, who wrote the *Scientific American* piece, suggests that the controversy may in part be evidence of paradigm clash. Mandelbrot is an applied mathematician and many of his critics are pure mathematicians, with different ideas about allocating credit for work. Horgan's suggestion places this coolness among Mandelbrot mathematicians in the context of a larger iceberg, which also includes the near-universal confusion in the public mind between the conduct and goals of pure and applied science, and between the ditto of science and engineering; it touches the glacial spread of university involvement in industry, the consequences of which nobody seems to be examining very closely. What it doesn't do is to warm the aforementioned coolness among Mandelbrot mathematicians.

Maybe Michael Spindler could explain to them about prima donnas and cooperation among competitors.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 12.**



MIDI, Turbo C++, Token Pasting, and PC Hot Keys

Al Stevens

I am writing this month from Redding, California, where the annual Redding Jazz Festival is underway. My other life often finds me at these events seated at a different, wider keyboard. This acoustic form of jazz that I know has not yet given way to technology. We use computers to keep track of bookings, income (what little there is), and such, but we do not play this style of music with machines. Other forms of music, including some newer forms of jazz, are using synthesizers, sequencers, samplers, and an interface called "MIDI" that computer manufacturers could learn from. I have no ear for the sounds and am certainly no expert in the technology, but it is fascinating. Jim Conger has written two books named *MIDI Sequencing in C* and *C Programming for MIDI*. Both are published by M&T Books (Redwood City, Calif.) and both address aspects of MIDI programming that a C programmer can use on a PC with a MIDI interface and Microsoft or Turbo C.

Conger explains what MIDI is and how it works, and he provides ample C code to achieve the effects he describes. I do not have any MIDI equipment, so I did not try to run the programs, but the text in the books is understandable and the code is readable, although the small, pale typeface that the publisher uses for code makes it hard to read. I used the diskettes to read the code instead. Maybe it's a way to get us to buy the diskettes, which anyone who gets a book for its code ought to do anyway.

The author says that musicians are being threatened by synthesized music. This is true. Many of my colleagues in music feel that threat; some of them have been touched by it. But Conger goes further and speculates that human composition, conducting, and performance can eventually be replaced by technology. He conjures an image of a digitized wire-frame model of Leonard Bernstein conducting a bank of synthesizers. Add an audience of rows and rows of nothing but digital audio tape recorders, and the picture would be complete.

Turbo C++

The news this month is Turbo C++. Following their success with object-oriented Pascal 5.5, Borland has assaulted the C++ object-oriented market with guns drawn and blazing. Originally intended to be an adjunct to Turbo C 3.0, the C++ face of Borland's compiler is now at its forefront. By emphasizing the object-oriented side of their most popular compiler product, Borland has made a policy commitment to object-oriented programming as the paradigmatic wave of tomorrow.

Included with Turbo C++ are a C++ compiler, an ANSI-conforming standard C compiler, and a completely overhauled Integrated Development Environment. The C++ compiler conforms to the AT&T 2.0 language specification, a standard of sorts but still a moving target. ANSI has launched its X3J16 C++ standards committee and we'll be hearing more about that undertaking in the months and years to come. Until then, purveyors of C++ language products have the AT&T reference manual and the AT&T implementation of C++ to aim at.

Most of the C++ programming systems available to PC users are ports of AT&T's cfront program, the translator that reads C++ source code and emits C language source code. The notable exception until now was Zortech C++, an implementation that is C++ 2.0 in spirit but that has numerous departures from the AT&T implementation. Borland has attempted to adhere closely to the AT&T reference manual.

I am writing this column in April, a month before the scheduled announcement of Turbo C++. You are reading it sometime near the middle of June, a month after the announcement, assuming no delays. There are always impediments to timely reporting when you work within this kind of schedule. My experience with Turbo C++ has been, therefore, gained only in beta-land. For that reason I can report no problems with the product. Sure, there have been problems, but this is a beta copy, and that's what a beta test is for.

I just finished writing a book called *Teach Yourself C++*. By the time you read this, Herb Schildt will no doubt be writing one with the same title. Don't be fooled by imitations. Mine has a yellow cover. (Sorry, Herb.) My book has about 130 exercise programs. Each exercise demonstrates a specific feature of the C++ language. As such, the book is something of a minor C++ torture test, and I used two ports of the AT&T cfront program to validate the exercises. Turbo C++, not a cfront port, passed with flying colors even in its beta configuration.

There are two different compilers in the product called Turbo C++. The other compiler is a full ANSI C compiler. Both compilers execute from within the Integrated Development Environment or from the command line. The program figures out what kind of program it is compiling based on the extension of the source file. A .C file is a C program. A .CPP file is a C++ program. The compiler needs to know the difference to know how to handle certain subtle differences in the languages. One of those differences involves "type-safe linkages."

Type-Safe Linkages

Type-safe means that a function's declaration and its callers use compatible types for the parameters. There has been a measure of type checking in C for a while. A C program specifies a function's return type and parameter list in the function prototype. If the function declaration or a call to the function does not match the prototype specification, the compiler declares an error. In the old days before prototypes a function was assumed to return an integer unless it was declared to return something else. You could pass it any number and types of parameters regardless of what it was expecting. The compiler did not care. The programmer had to wrestle with the bugs that resulted when the program passed one thing and the function expected something else. C++ needs stronger typing and so its designer created the function prototype, which was adopted by many C compilers and eventually by the ANSI C standards committee. The prototype allows the compiler to insure that calls to a function match the function's declaration with respect to its return type and parameter list.

The C prototype is an effective measure as long as the source file that declares the function and the one that calls it are using the same copy of the prototype. By convention, most programmers will put the prototype in a header file and include the header in the source file that declares the function and all others that call the function. But this practice is only a convention and nothing in the C language enforces it. If you use different prototypes for the same function in different source files, the calls to the function can have different configurations with the same undesirable results that we had before prototypes. The problem is, of course, that you compile the source files independently, and the linker has no way to reconcile different function prototypes between independently compiled object modules. The traditional method for controlling this situation is to use the header file convention mentioned above and a well-defined make file that assures that all affected source files compile when a header file they include changes.

Most of the C++ programming systems available to PC users are ports of AT&T's cfront program, the translator that reads C++ source code and emits C language source code

C++ requires stronger type checking across linked modules because much of the integrity of a program depends upon the correct use of overloaded functions. Therefore, C++ 2.0 introduces a feature called "type-safe linkage" to guarantee that function prototypes are consistent across independently compiled source files. To achieve this guarantee, the C++ compiler translates function names into "mangled" names. The translator adds characters to the function name to encode the parameter list. For example, the following function prototype would generate the following mangled function name.

```
long foo(int, char *, struct tm);
// mangled name: _foo__FiPc2tm
```

The mangled name specifies, in cryptic notation, the types of the parame-

(continued from page 132)
ters. An external function does not, therefore, have the name you gave it. If you prototype it differently in a remote source file, the mangled name will be different, and you will get unresolved symbols when you link the program. Thus you have type-safe linkage at the parameter list level. For some reason, this technique does not similarly encode the function's return type.

These mangled names are usually invisible to the programmer unless you read the linkage memory map, but you must know about them because the unresolved symbol errors will report the mangled names rather than the ones you lovingly bestowed and you need to find the original name among the mangled mess. There will be times when you need to tell C++ not to mangle.

Suppose your C++ program calls a function that was compiled by a C compiler. That is not an unusual thing to do. The entire standard C function library is available to C++ programs. So, unless you tell the C++ compiler otherwise, it will mangle the function name in the call, and the linker will not find a matching function. Therefore, the C++ compiler needs a way to know that a named function was compiled by a C compiler. C++ uses this construct to achieve that purpose.

```
extern "C"
{
// C things ...
}
```

You can put prototypes between the curly braces, and the compiler knows that calls to those functions must be made without mangled names. C++ compilers usually include standard C header files that already have the extern "C" statement built in.

If you surround an entire function with the extern "C" braces, that function will compile with C linkages. You would need to do that in the case where you are using a C library function that requires your program to provide a named function. Of course, such functions do not enjoy the same type-safe linkage that regular C++ functions have.

ANSI C Token Pasting

A few months ago I wondered about the ANSI C preprocessor ## token pasting operation. If you code the ## operator in a macro replacement list, it replaces the two parameters that surround it by "pasting" the two arguments together into one argument and then continuing with preprocessing. For example:

```
#define pasteup(a,b) a ## b
foo(pasteup(12, 34));
```

The resulting statement would be:

```
foo(1234);
```

One is hard pressed to imagine a use for this feature. Necessity being the mother she is, we can only speculate that the invention of ## sprang from the loins of need. The behavior of ## is more peculiar when you consider this:

```
pasteup(hello,dolly)( );
```

The effective statement is:

```
hellodolly( );
```

which makes us suspect that ## could be of some use in the preprocessor fabrication of identifiers.

The ## behavior is well documented in the ANSI C standard, but there are no good examples of its use or clues to its purpose. That bothered me because I couldn't come up with a good use for it. So, as an experiment, I went looking for an application for the odd little critter. It would have been unsporting to call someone on the X3J11 committee and ask what it was for but I wanted to find a real use for token pasting, and eventually I did. What I

found, however, was not code that used the ANSI token paster at all. In fact, what I found was not in a C program. Instead, I found a contrivance in the generic.h header file that comes with C++ 2.0 compilers. The following construct works with pre-ANSI C and C++ preprocessor programs:

```
#define name2(a,b) a\
b
```

The purpose of this macro and others like it are to paste type and class names together to form what has been called a "parameterized" class from a generic one. In *Programming in C++*, Prentice Hall, 1989, Dewhurst and Stark describe what they call "genericity," still another abominable non-word that someone, out of generosity and in the true spirit of computer literature, has contributed to the English-speaking world. There they observe that ANSI-conforming compilers could achieve the same thing with this:

```
#define name2(a,b) a##b
```

Eureka! Now a quick look at the generic.h of Turbo C++ reveals that they do indeed use the ## operator in just that fashion and my quest is complete.

My original comments about ## came in a review of Rex Jaeschke's book, *Mastering Standard C*. Some readers called to say that they could not locate the book or the publisher. Here is the address: Professional Press, 101 Witmer Rd., Horsham, PA 19044, 215-957-1500.

The C++ dialect is destined to replace C as the language of choice for developers of software systems

The Future of C

Turbo C++ marks the dawn of an era. C++ proponents would say that the era started a long time ago, but from my view, the era began when a well-established and leading language vendor such as Borland bet the farm and came out full-bore with a complete C++ development environment. You can bet that Microsoft is not far behind.

The C++ dialect is destined to replace C as the language of choice for developers of software systems. The combination of objects, user-defined data types (same thing, only different), and the C syntax makes C++ a natural heir to the throne. But there is at least one category of program that will never switch to C++. Developers of utility and systems programs will stick with C for its freedom and close-to-the-metal feel and for its lean and mean executables. C++ programs can have those qualities, but at the expense of extensibility. Add all those extensible data types and your programs will spend time and memory with layers of constructors and destructors and class hierarchies. Leave them out and you are using a C++ compiler to develop a C program. C lives.

By the way, if you want Turbo C without C++, you can still buy Turbo C 2.0. Borland will continue to market and support that version of the compiler as the C language entry-level compiler to compete with Microsoft QuickC. I do not know prices just now, but you should expect Turbo C 2.0 to be a bit cheaper than when it was this year's snappy model. I would have preferred to see a TC 3.0 that fixed the ANSI incompatibilities and the bugs and used the new, improved IDE, but apparently

Borland decided to invest their resources in the C++ package and leave 2.0 the way it is for those who want it.

Hot Keys

We've all used programs that employ hot keys. On the PC, hot keys are key combinations that a program uses to cause things to happen. The most typical use is to make a TSR pop up. By definition, a hot key should be a key combination that applications do not use in their normal command set. By convention, hot keys are combinations that do not produce anything on the screen in normal DOS use and that are not one of the generic function keys. Sidekick uses the Ctrl-Alt combination. Ready uses the 5 on the numeric keypad (with NumLock off).

C++ requires stronger type checking across linked modules because much of the integrity of a program depends upon the correct use of overloaded functions

A programmer has three problems to solve with respect to hot keys. First, how do you read one from the keyboard to let the user change the hot key setting? Second, how do you translate that into a meaningful display to tell the user what the key combination is? Third, how do you know when the hot key has been pressed? The code that accompanies this month's column addresses those problems.

Listing One, page 146, is hotkey.h, a header file that an application program will include to use the hot key functions. It contains the prototypes.

Listing Two, page 146, is testhk.c, a program that demonstrates the use of the hot key functions. Hot keys are described in terms of their keyboard scan code and the BIOS shift key mask. Each key on the keyboard sends a unique scan code to the computer when you press a key. These codes are not ASCII, and you will find them documented in many books about programming the PC. The BIOS shift key mask is a byte in low RAM that represents the current status of the Shift, Ctrl, and Alt keys. The hot key software takes care

of the translation of the codes to the key values.

The testhk.c program begins by displaying these messages to describe what it expects you to do:

```
Press a hot key
Press Esc to keep the one you have
Press the Space Bar to clear the one
    you have
Press Enter to use the new one
```

Then it calls the *hotkey* function, passing the addresses of an integer for the scan code and one for the shift key mask. The *hotkey* function lets the user press different hot key combinations, displaying each one until the user presses Enter, Esc, or the Spacebar.

Once the user has selected a hot key, the testhk program displays its value by calling the *showkey* function, which displays something like Ctrl-X on the screen. Then testhk calls *initkey* to initialize testing for the hot key. The *iskeyhit* function returns nonzero if the hot key has been pressed and zero if not. The testhk program loops waiting for a press of the hot key. Then it calls *endkey*.

It is important that the program calls *initkey* before testing for the hot key press and *endkey* afterwards. Don't leave them out.

Listing Three, page 146, is hotkey.c, the functions that manage hot key programming for your application. The *initkey* function attaches the keyboard interrupt vector to the *newkb* interrupt service routine. That function simply reads the keyboard input port into a variable named *kbval* and chains to the old interrupt vector. On some PCs you do not need the interrupt service routine. Anywhere the program examines the value in *kbval* you could simply read the keyboard input port 0x60. Other computers do not work that way. The input port delivers a meaningful value only after an interrupt 9. Therefore, the program attaches to interrupt vector 9 to read the value of the keyboard port.

The *iskeyhit* function accepts a scan code and shift key mask and tests them against the current values of *kbval* and the BIOS shift key mask. The function returns true if the values match and false otherwise.

The *hotkey* function programs a hot key. When the user presses a valid hot key combination, the function displays an ASCII representation of it on the screen and waits for another key press. When the user presses Enter, the scan and shift mask values associated with the most recent key are returned to the caller. If the user presses the Esc key, the caller's original values are returned. If the user presses the Spacebar, zeros are returned.

Reading a hot key is no cake walk. Neither DOS nor BIOS return anything when you press one of the key combinations that we have said are valid hot keys. What you must do is examine the input from the keyboard port and react according to the scan codes that come in. The first thing to do is wait until the user gets off the keyboard. The most significant bit (0x80) of the scan code will be zero if any key is being held down. Otherwise it will be one. Next, you must clear the BIOS read-ahead buffer and keep it clear. Every time the user presses a valid BIOS key, BIOS adds an entry to a circular read-ahead buffer. By clearing that buffer, you prevent BIOS from beeping when the buffer is full. Then you go into a loop reading the keyboard port waiting for a key to be pressed. If that key is the scan code for the Esc, Enter, or Spacebar key, you are done. Otherwise, it must be the Ctrl, Alt, or a Shift key. After that you wait for another key to be pressed. Then you wait for a third key press or a key release at which time you have the two or three scan codes that constitute the hot key. From them it is a simple matter to validate the combination.

The user might try to program an invalid hot key. You would not want to use the letter A or the Home key as hot keys, for example. A hot key must be a combination that includes one or more of the Shift, Ctrl, and Alt keys and another key. Or a valid hot key can be any two or more of the Shift, Ctrl, and Alt keys. If the user selects anything else, the *hotkey* program buzzes and rejects the selection.

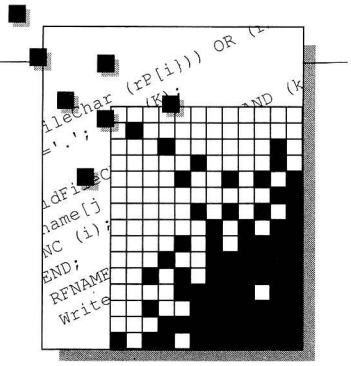
The *showkey* function returns a pointer to a display string that it constructs from its scan code and shift key mask parameters. You can use this string to display the name of the hot key.

These functions compile with Turbo C. To use them with another compiler, you must use that compiler's version of the *setvect*, *getvect*, *sound*, *delay*, and *nosound* functions. Most compilers have variations on the first two. Microsoft C uses *_dos_getvect* and *_dos_setvect*, for example. The other three are Turbo C specific. You can use whatever method your compiler has to make a noise through the speaker or you can stub out the buzz function.

DDJ

(Listings begin on page 146.)

Vote for your favorite feature/article.
Circle Reader Service **No. 13.**



The Just One Thing Dilemma

Jeff Duntemann KI6RA/7

Arizona! Carol and I rolled into Phoenix in the middle of a furious windstorm, packed into the Magic Van along with dogs, assorted succulents, and a truncated ficus, as well as anything Allied Van Lines had forgotten to pack the previous Wednesday.

Rolling clouds of trendy Santa-Fe style dust were blowing off the parched land and making visibility difficult in the Safeway Parking lot somewhere near Bell Road and 19th Avenue. I was picking my way carefully through the gloom when something large and brown bounded in out of nowhere, caromed off a Jeep Wrangler and crossed the lane immediately in front of me at considerable speed. I hit the brakes and watched as it wedged itself solidly under a *USA Today* newspaper rack.

"I bet that's a tumbleweed!" Carol said with some excitement.

"But this is a supermarket parking lot!" I objected, feeling all the romanticism of the Wild, Wild West, including images of singing cowboys, tumbling tumbleweeds, and wild coyotes (now made mostly of marine plywood and sold in Hallmark stores) take a sharp turn south and vanish into the Nostalgia Zone.

These days, I guess one escapes from dust storms by taking refuge in the nearest Circle-K. The cowboys all wear feed caps and listen to rap tapes in their jacked-up Ranchero 4×4s. But by gully, those tumbleweeds have refused to sell out — even if it means tumbling right in the front door of K-Mart.

The world — and Phoenix — are full of surprises. What's true of Phoenix is true of the structured programming world as well. I like surprises, and there have been a bunch of them in recent months.

The most striking is the sudden appearance of three brand new Pascal compilers. I'd gotten used to thinking of Turbo Pascal as unassailable in the marketplace when QuickPascal appeared and did quite well, thank you. I then assumed that two major Pascal compilers was a mighty crowded field — and now, perhaps, I may be wrong in that as well.

I've just made contact with the QuickByte Pascal people, so we'll push them on the stack until a future column. The other two Pascal products come from Jensen Partners and Stony Brook. I'll be testing them thoroughly in coming months, and you'll get a report when the testing's done.

Crossing a Banana with a Gorilla

JPI's new Pascal is notable for supporting objects with multiple inheritance. Multiple inheritance is one of those sleeper concepts that (in my opinion) nobody in this business quite understands as yet. Intuition tells me that it represents a radically different way of thinking about object design. I'll know when I've spent some time with it — and I'll be asking some largish questions including whether it is in fact a good thing at all. In the meantime, let me take a little time to explain what multiple inheritance means.

In single-inheritance systems such as Smalltalk, Actor, and Turbo/QuickPascal, a class (object type in Turbo Pascal jargon) may have only one parent class. Pretty obviously, multiple inheritance allows a class to inherit from more than one parent class.

Think for a bit about that catering truck that blows into your office complex parking lot at 10 A.M. sharp, most appropriately playing "La Cucaracha" at 70 dB on its electronic horn. A catering truck is a truck: It has wheels, an engine, a steering mechanism, and a power train. A catering truck, however, is also a kitchen: It has a stove, a sink, a refrigerator, cabinets full of food, and a microwave oven. In the grand object hierarchy describing our wonderfully surprising world, the class *CateringTruck* inherits from both *Truck* and *Kitchen*.

Now, we've been talking about objects for most of a year as though any given class can only be a more specific offshoot of its parent class. A menu is a specific kind of window, which is a specific kind of rectangular area on the screen, and so forth. A menu may touch on many areas of the system at large, allowing you to choose files, or parallel ports, or customer numbers. Nonetheless, at the heart of it a menu is a kind of window. It pops up, does its thing, and then vanishes.

Applying that logic to class *CateringTruck* is not so clean. In a single-inheritance system, you have to make a choice: Is *CateringTruck* fundamentally a truck or a kitchen?

You Choose, You Lose

OK, let's choose, and draw it all out in Figure 1. If pressed, I'd say *CateringTruck* was more fundamentally a truck than a kitchen, so we'd place *CateringTruck* beneath *Truck* in the object hierarchy. *Truck*, in turn, is a child class of the *SelfPropelledVehicle*, which in turn is a child of base class *Vehicle*, as is *RailroadCar*. *Vehicle* defines those few methods common to all vehicles, such as *Stop*. (The expected *Start* method is not defined until the *SelfPropelledVehicle* class. A truck can put itself in gear and move forward . . . but a railroad car can only put on its brakes.)

Kitchen is defined in a separate small hierarchy and is the child of base class *ChemLab*. (Think about it . . .) *Kitchen*'s methods include *WashDishes*, *PrepareNextMeal*, and so on.

Adding kitchen capabilities to vehicles is done by adding objects of class *Kitchen* to the definitions of *CateringTruck* and *DiningCar* as ordinary fields, in the same way that you would add an integer or a string field. The fact that *Kitchen* is an object class is incidental. The notation for invoking the *WashDishes* method would be *ADiningCar.Kitchen.WashDishes* or *ACateringTruck.Kitchen.Washdishes*. Now, is this such a bad thing? The vehicles still roll. The kitchens still cook. But . . . polymorphism only operates along the line of inheritance.

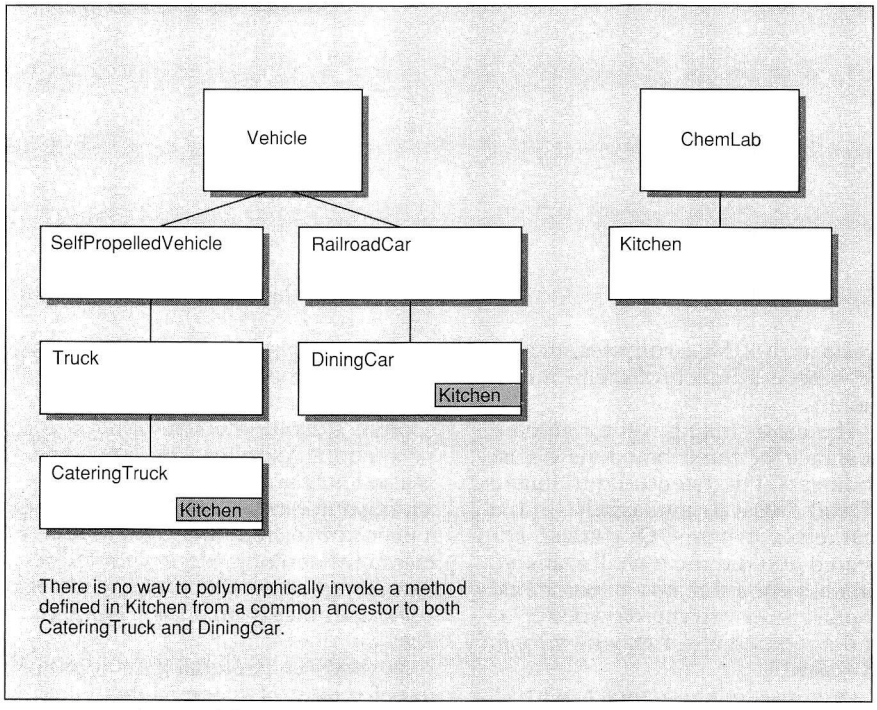


Figure 1: *The need for multiple inheritance*

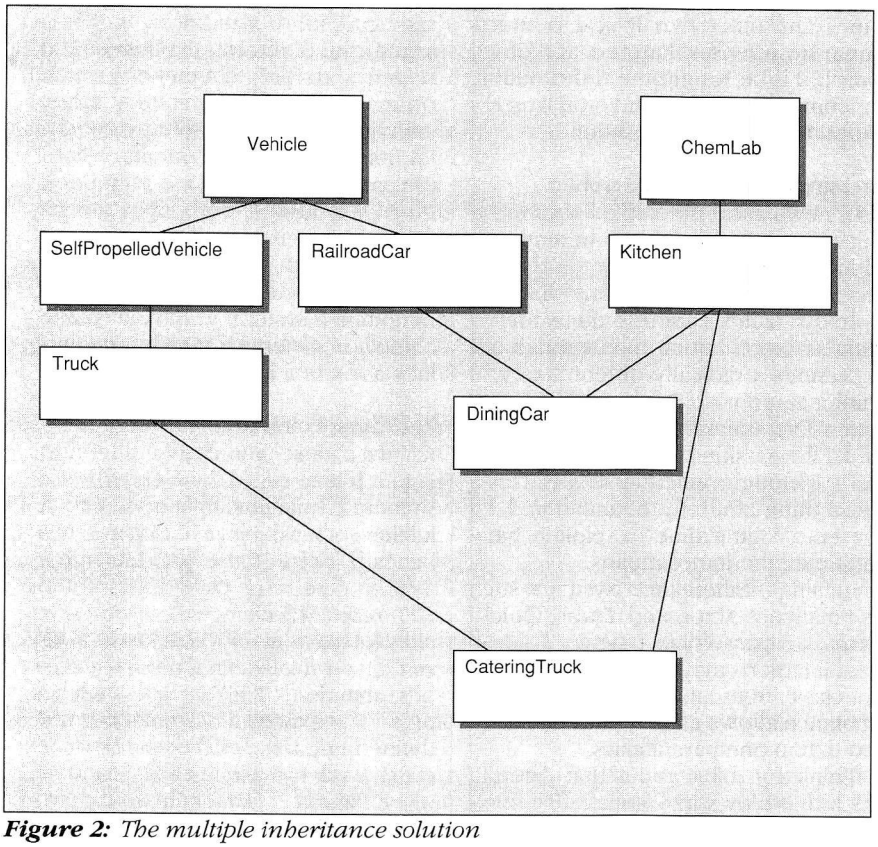


Figure 2: *The multiple inheritance solution*

What this means is that, whereas you could invoke a method like *StopMotion* on any child class of *Vehicle*, you could not invoke the *PrepareNextMeal* method belonging to *Kitchen* upon an object accessed polymorphically through a pointer to class *Vehicle*, even though both *CateringTruck* and *DiningCar* descend from *Vehicle*. The problem is that no common ancestor of *CateringTruck* and *DiningCar* contains the *Kitchen* object — certainly not *Vehicle*.

We could, of course, add the *Kitchen* field to the definition of type *Vehicle*. That, however, defeats the whole purpose of object-oriented design: To distribute functionality across an object hierarchy as appropriate. All vehicles do not contain kitchens, so forcing a kitchen field into the *Vehicle* class, while syntactically correct, is semantically absurd.

The Just One Thing Dilemma

What we're seeing here is a problem I'm calling the "Just One Thing dilemma." If object-oriented design is a process of modelling some portion of reality in logical abstractions, then single-inheritance systems impose an artificial limitation on the modelling process: The assumption that all objects are fundamentally Just One Thing — that one aspect of a modelled object is necessarily deeper than all others, and that that aspect is the only one through which the polymorphism mechanism operates.

Still not convinced that this is a problem? Let's return to the Metaphor Zone for a moment. Suppose we're modelling a major sports event in a stadium in Baltimore. Some people are coming down from New York by train, and will arrive just before the game begins. Others are driving in, and will be milling around in the parking lots before game time. Everybody needs to be fed in time to be finished before the game begins. So at 5:00 sharp, we issue a command to all vehicles capable of providing food: *PrepareNextMeal*. Each type of vehicle responds in its own appropriate fashion, according to the principles of polymorphism. The porters in the Metroliner begin preparing quiche in the dining car, and the owners of the fleet of catering trucks cruising the stadium parking lots begin putting hot dogs on to grill. One way or another, everybody waiting to get in to see the big game gets fed — and all at the direction of a single command common to all food providers.

With single inheritance, there is no

(continued from page 140)
way to issue that single *PrepareNextMeal* command to both the catering trucks and the dining cars. We have decided that catering trucks and dining cars are fundamentally vehicles, and the only aspects the two have in common are those inherited from their common ancestor, *Vehicle*. The fact that both contain a field named *Kitchen* is incidental. They did not inherit their kitchenness — it was just sort of glued on after the fact. And polymorphism operates only through inheritance.

In most implementations, whereas colliding identifiers are made invisible, they still exist within an object

If object-oriented design is to be a modeling process, then we have to face up to the fact that real-world objects are often mongrels with conceptual parentages going off in every direction at once. The more complex the system, the less likely that any given element in the system will be Just One Thing.

The Multiple Inheritance Solution

The solution is to let dining cars and catering trucks inherit their vehicleness from class *Vehicle*, and their kitchenness from class *Kitchen*. Figure 2 shows the classes of Figure 1 redrawn to indicate this new set of relationships. *DiningCar* and *CateringTruck* are now equally descended from *Vehicle* and from *Kitchen*. Each has everything that a *Kitchen* object has, and each has everything that a *Vehicle* object has.

Polymorphic method calls are no longer a problem. *CateringTruck* can be addressed as easily as a *Vehicle* object as it can be addressed as a *Kitchen* object. To direct all food service vehicles to stop their motion, you gather the food service vehicles into a linked list of *Vehicle* objects and simply call *Vehicle*. Stop for each vehicle on the list.

On the other hand, to direct all food service vehicles to prepare the next meal, you gather the same food service vehicles into a linked list of *Kitchen* objects instead — and then call the *PrepareNextMeal* method for every object on the list.

When Fields Collide

One major semantic problem with multiple inheritance occurs when a class inherits the same identifier from more than one parent. For example, a *Vehicle* object might have a data field called *Door* — and so might *Kitchen*, since the two concepts we're modelling, vehicles and kitchens, both have doors. So if you want to call the method *MyCateringTruck.OpenDoor*, which method is called: The *OpenDoor* method inherited from *Kitchen*, or the *OpenDoor* method inherited from *Vehicle*? Or, for that matter, which *Door* field is acted upon?

Different languages treat this problem in slightly different ways, but in general, when two inherited identifiers collide, *both* are rendered invisible within the scope of the class that inherits them. In other words, when it inherits two different *Door* fields, *CateringTruck* objects lose the ability to manipulate either.

Then again, if the *Kitchen* stove catches fire, you're going to want to manipulate the door in one helluva hurry. *CateringTruck* objects may legitimately have doors. The question you the designer must answer is, which door field will be used within *CateringTruck*? Your implementation of the *CateringTruck* object might be such that the doors are nothing more than the same doors defined in the *Vehicle* object. You must then explicitly specify that the *Door* field that will be visible within *CateringTruck* will be the *Door* field inherited from *Vehicle*.

This can be done in many ways, and I'll explain how it's done in TopSpeed Pascal once I actually receive a copy of the product. (At this writing I have only seen the documentation.) The important thing to understand is that you the designer make the choice, at compile time, which of two colliding identifiers will remain visible.

Weaving a Tangled Web

A lot of researchers are uncomfortable with multiple inheritance because it can easily become a creator of far more complexity than it manages as a structured tool. The simple metaphor of the *CateringTruck* and *DiningCar* classes works tolerably well because there is no more connection between two otherwise independent object hierarchies than there must be to do the job. It doesn't take a lot of imagination to see what might happen when 7 or 8 or 20 complicated object hierarchies begin mixing it up in a big way.

In most implementations, whereas colliding identifiers are made invisible, they still exist within an object, taking up memory even if they are not accessible. In other words, if a class inherits six colliding identifiers called *Door* and only uses one of them, you still have five invisible (and useless) *Door* fields inside every instance of that class. (I think there is a mechanism built into C++ 2.0 to ensure that only one of a set of colliding members is physically present within an object instance, but it is a measure of the difficulty of C++ that I can't really tell as yet.)

I suspect that over time a new kind of coupling will be recognized in OOP design: The degree to which two or more object hierarchies are interconnected via multiple inheritance. I further suspect that less will be more here, both to keep object instances small and efficient, and also (at a higher level) to keep the whole conceptual nature of the object hierarchy from turning into something the color of Arizona mud. And while a given concept being modeled as an object may be more than Just One Thing, it is just as certainly not Ten or Twelve Things, either.

Remember: Moderation in all things . . . er . . . objects.

Out of Sight, Out of Mind

Of course, it's easy to talk about catering trucks and dining cars and things that bear no relationship to the situations we encounter as DOS programmers, right? Well, let's bring the multiple inheritance question a little closer to home, by asking, "What is a menu, really?"

An Ivory Soap majority (99.44/100%) would say that a menu is a screen mechanism through which you choose one item from a displayed list of several. The underlying assumption is that the screen mechanism is the essential aspect of the menu. Data? What's data? Oh . . . that stuff. . . .

I guess it's human nature to assume that what is visible is most important. But consider: There would still be data without menus, but a menu without data would be useless and futile. Let's move data back to the true center of things and define a menu as an array whose index is directly controlled by the user through the keyboard and screen. You could also call it a list whose current pointer is directly controlled by the user, depending on the implementation. (Let's take a step back from the nature of the implementation and call any class that consists of more than one data item a *collection.*)

This isn't sophistry, but an issue that strikes to the heart of object-oriented design. Writing a menu object and passing it an array or list as a parameter is the Old Way. In keeping with the object metaphor of active data, each collection class should contain its own menuing method. In other words, when we want to choose one item out of a collection named *MyCollection*, we should be able to make a call something like this:

```
ChosenItem := MyCollection.Choose;
```

To get you comfortable with this notion at the source-code level, I've written a very simple demonstration unit for Listing One (ARROBJ.PAS), page 149, that implements a string array class in Turbo Pascal 5.5. Class *StringArray* follows the strict interpretation of object-orientation: That a class' internal fields are private and may be accessed only through function calls. This imposes a certain performance hit, but in return it allows the physical implementation of internal data to be anything at all. For example, you could actually store the elements of *StringArray*'s "array" as records in a random-access file, with the index of the array acting as the random file pointer. Obviously this makes access time dependent on the disk speed, but would allow you 32,767 elements of any reasonable size — something you

simply can't do when limited by Turbo Pascal's single 64K data segment.

(Note that other changes would be required to support an array larger than 15 items or so, because in this simple implementation there is no scrolling logic, and the entire array must be small enough to be displayed on the screen at once.)

Listing Two (page 150) is a simple demonstration program that creates an instance of *StringArray*, fills it with strings of random characters, and then invokes the *Choose* method to display the built-in menu and allows the user to choose one of the 15 elements in *MyArray*. There is also a *Display* method that simply puts the contents of the array up in a window for inspection, without any opportunity to select and return an individual element.

I guess it's human nature to assume that what is visible is most important

The important thing to understand about the *StringArray* class is that it is first and foremost an array, to be used where an array should be used. The menuing mechanism is a convenience, and can simply be ignored if you don't happen to need a menu.

Not My Yob, Mon

In my view of an object-oriented world, all collection classes should contain both browsing and menuing methods. In very simple classes such as *StringArray*, this doesn't involve a lot of complicated code. On the other hand, in an application written around an elaborate text windowing scheme, having collection classes implement their own menuing and browsing code is wasteful and redundant.

So who should make the menu? In an ambitious application with an object-oriented user interface, making menus is not the job of a collection class. Worse, when important polymorphic method calls must be made to all active windows (like *ReDraw*) a menu must be descended from the abstract window class, which defines the *ReDraw* method. On the other hand, gluing a collection class onto a menu object as an ordinary field forbids polymorphic access to the menuing method from a collection class ancestor.

This is where multiple inheritance earns its keep: By allowing object hierarchies to specialize without forcing object classes into unnatural unions — like making collection classes descendants of screen window classes. In a multiple-inheritance environment, the collection classes inherit fundamental menuing methods from the user-interface hierarchy without losing their true heritage as collections.

We'll See

Needless to say, this is all background information. In a future column I'll give you some real code in TopSpeed Pascal to show you how multiple inheritance works. As I've tried to show here, the concept seems to solve a couple of really knotty reservations I've had about objects all along. This isn't to say it won't uncover half a dozen more. (Life is like that.) I'll let you know as soon as I know myself.

Odd Lots

Traffic from earlier columns has been piling up during our mad dash to the desert. Let's read the mail. . . .

First of all, I need to back away from an assertion I made last year about Smalltalk/V PM. The way it was explained to me, Smalltalk is almost by definition an interpreted system, and while the interpreter can be hidden inside a "sealed off" application, it's still gotta be in there somewhere. The explanation is complicated, but suffice it to say that Jim Anderson and his crew of wizards have shown me to my own satisfaction that they have in fact made a compiling Smalltalk that loses nothing functional — and gains them an immense performance edge over interpreted implementations. Sooner or later, when I find a version of OS/2 PM that will make friends with my machine, I'll probe further.

And while I'm backing away from that one, I'll keep going and admit that my gripes about Modula-2's limited set size are way behind the reality of today's commercial compilers, as numerous readers have pointed out. In fact, with the TopSpeed and Stony Brook compilers, not only can you have SET OF CHAR, but you can have a set of CARDINAL — with up to 65,536 elements. If anything, Wirth erred here by giving compiler implementors too much leeway to decide the size of sets. Had he said flat-out up front that sets must support 256 or 65,536 elements, we'd have a less ambiguous standard and I'd have less opportunity to be wrong.

Several people have pointed out in letters that TopSpeed Modula-2 adds two operators to the language for bit-shifting. The >> operator shifts to the right and << shifts to the left. Missed that somehow, but it's an extension well worth having.

Steve McMahon wrote to warn against using global variables in Modula-2 modules (as I did in BITWISE.MOD, presented in March 1990) whenever possible. Modula-2 is a multitasking language, and global variables make modules pointedly non-reentrant, since the interruptor can futz a global that the interruptee isn't through with yet. Hoo-boy; hadn't thought of that one. Fortunately, my understanding of reentrancy indicates that making both variables local to the procedures in which they are used will solve the problem, because each entry of a procedure creates its own stack frame, where local variables live their short lives.

I've very recently received a damned fine Modula-2 communications library from a very small company in Ohio. Solid Software's Solid Link supports interrupt-driven communications via COM1 through COM4, with buffered input and output. It also has the distinction of being the only toolbox on my shelf for any language that supports the notoriously difficult-to-implement ZModem file transfer protocol. (It also supports XModem, YModem, YModem-G, TeLink and SEALink, plus some variations on the above.) The library is implemented entirely in assembler, and claims to be able to communicate at 19,200 baud or faster. (I haven't done any speed tests as of yet.)

The manual is fairly terse, but it has the virtue of containing numerous code examples including a complete and thoroughly nontrivial terminal program. Certainly if you need to implement any file transfer protocols in Modula, you can save yourself a titanic amount of work by letting someone else do the hard stuff. Highly recommended.

Products Mentioned

TopSpeed Pascal
Jensen & Partners International
1101 San Antonio Rd., Ste. 301
Mountain View, CA 94043
415-967-3200
Price: Contact Vendor

Solid Link
Solid Software Inc.
P.O. Box 8132
West Chester, OH 45069
513-777-1414
Price: $199

Plenty of Here Here

I believe it was Gertrude Stein who said of Oakland, "There's no there there." Not so — the real problem with California is that there's not enough There to go around. She should have tried Arizona. There's so much here here that that won't be a problem for some time to come. The sky seems so enormous as to swallow you whole, out on the high desert north of Cave Creek, where I expect to live someday. People warn me of those days when it gets down to 108 degrees after midnight. I guess. But then again, you don't have to shovel heat out of your driveway. In the meantime, Mr. Byte has learned that you can in fact lift your leg on a saguaro cactus . . . if you're careful.

Maybe twelve years of wandering are enough.

Maybe I've come home.

Contact Jeff Duntemann on MCI Mail as JDuntemann, or on CompuServe through ID 76117,1426.

DDJ

(Listings begin on page 149.)

Vote for your favorite feature/article.
Circle Reader Service **No. 14.**

Listing One *(Text begins on page 131.)*

```
/* ---------- hotkey.h ---------- */

char *showkey(unsigned keyscan, unsigned keymsk);
int iskeyhit(char keyscan, char keymsk);
void hotkey(unsigned *keyscan, unsigned *keymsk);
void initkey(void);
void endkey(void);
```

End Listing One

Listing Two

```
/* ------------- testhk.c --------------- */

/*
 * A program to demonstrate the use of the hotkey programmer
 */

#include <stdio.h>
#include "hotkey.h"

void main()
{
    unsigned scan = 0;
    unsigned mask = 0;

    printf("\nPress a hot key");
    printf("\nPress Esc to keep the one you have");
    printf("\nPress the Space Bar to clear the one you have");
    printf("\nPress Enter to use the new one\n");

    /* ------- program the hot key ---------- */
    hotkey(&scan, &mask);

    if (scan || mask)    {
        /* ------- display the programmed hot key --------- */
        printf("\nThe new hot key is %s (%02x, %02x)\n",
            showkey(scan, mask), scan, mask);

        initkey();
        while (!iskeyhit(scan, mask))
            printf("\rWaiting for you to press the hot key..");
        endkey();

        printf("\n%s was pressed", showkey(scan, mask));
    }
    else
        printf("\nNo hot key programmed");
}
```

End Listing Two

Listing Three

```
/* ---------- hotkey.c ------------ */

#include <stdio.h>
#include <conio.h>
#include <bios.h>
#include <dos.h>
#include <string.h>
#include "hotkey.h"

#define KYBRD 9

/*
 * Program a hot key
 */

static char hotky[50];

static int multibits(int k);
static void clearkb(void);
static void buzz(void);

char *scodes[] = {
    "Esc",
    "1", "2", "3", "4", "5", "6", "7", "8", "9", "0",
    "[-]",
    "[=]",
    "Bksp",
    "Tab",
    "Q", "W", "E", "R", "T", "Y", "U", "I", "O", "P", "[", "]",
    "Enter",
    "Ctrl",
    "A", "S", "D", "F", "G", "H", "J", "K", "L",
    "[;]",
    "[\"]",
    "[`]",
    "LShift",
    "\\",
    "Z", "X", "C", "V", "B", "N", "M",
    "[,]",
    "[.]",
    "[/]",
    "RShift",
    "PrtSc",
    "Alt",
    "",
    "",
    "F1", "F2", "F3", "F4", "F5", "F6", "F7", "F8", "F9", "F10",
    "",
    "",
    "Home",
    "Up",
    "PgUp",
    "[-]",
    "[<-]",
    "[5]",
    "[->]",
    "[+]",
    "End",
    "Dn",
    "PgDn",
    "Ins",
    "Del"
};

static void (interrupt *oldkb)(void);
static void interrupt newkb(void);
static int kbval = 0;

/* ----- keyboard ISR ------ */
static void interrupt newkb(void)
{
    kbval = inportb(0x60);
    (*oldkb)();
}

/* ------------ initialize for hot key test ----------- */
void initkey(void)
{
    /* -------- attach to keyboard interrupt -------- */
    oldkb = getvect(KYBRD);
    setvect(KYBRD, newkb);
}

void endkey(void)
{
    setvect(KYBRD, oldkb);
}

/* --------- test for a specified keystroke ----------- */
int iskeyhit(char keyscan, char keymsk)
{
    char far *bp = MK_FP(0, 0x417);
    int rtn;

    /* -------- if no key is depressed, bail out ---------- */
    if ((kbval & 0x80) != 0)
        rtn = 0;
    else if (keyscan && kbval != keyscan)
        rtn = 0;
    else
        rtn = (keymsk == ((*bp) & 0xf));

    return rtn;
}

/* ---------- program the user hot key ----------- */
void hotkey(unsigned *keyscan, unsigned *keymsk)
{
    int key1, key2 = 0, key3, k = 0;
    char svbios;
    char far *bp = MK_FP(0, 0x417);

    /* ---------- attach to keyboard interrupt ------------- */
    initkey();

    /* ------- first, we need to be off the keyboard ------- */
    kbval = 0x80;
    while ((kbval & 0x80) == 0)
        ;

    /* --------- clear the BIOS readahead buffer ---------- */
    clearkb();
    svbios = *bp;
    while (1)    {
        key1 = 0x80;
        key3 = 0;
        /* ------- wait for a key press --------- */
        while (key1 & 0x80)
            key1 = kbval;
        /* ------ if Spacebar, Enter or Esc, drop out ------ */
        if (key1 == 57 || key1 == 1 || key1 == 28)
            break;
        /* ------ must be Alt, Ctrl, or Shift -------- */
        if (key1 != 56 && key1 != 29 &&
            key1 != 42 && key1 != 54)    {
            buzz();
            continue;
        }
        /* ------- wait for a second key press -------- */
        while ((key2 = kbval) == key1)
            if (key2 & 0x80)
                break;
        key2 &= 0x7f;
        k = 0;
        if (key1 == key2)
            /* ----- released the first key ----*/
            continue;
        /* ---- wait for key release or key change ----- */
        while (((key3 = kbval) & 0x80) == 0)
            if (key3 != key2 && key3 != key1)
                break;
        key3 &= 0x7f;
        if (key3 == key2 || key3 == key1)
            key3 = 0;
        /* -------- clear the bios readahead buffer -------- */
        clearkb();
        *bp = svbios;
```

```
        /* ------ look at the keystrokes -------- */
        switch (key1)    {
            case 56:    k |= 8; break;
            case 29:    k |= 4; break;
            case 42:    k |= 2; break;
            case 54:    k |= 1; break;
            default:    break;
        }
        switch (key2)    {
            case 56:    k |= 8; key2 = 0; break;
            case 29:    k |= 4; key2 = 0; break;
            case 42:    k |= 2; key2 = 0; break;
            case 54:    k |= 1; key2 = 0; break;
            default:    break;
        }
        switch (key3)    {
            case 56:    k |= 8; key3 = 0; break;
            case 29:    k |= 4; key3 = 0; break;
            case 42:    k |= 2; key3 = 0; break;
            case 54:    k |= 1; key3 = 0; break;
            default:    break;
        }
        if (key2 != 1 && key2 != 28)    {
            if (key2 == 70 || key3 == 70)    {
                /* ---- can't use the Break key ---- */
                buzz();
            }
            else    {
                if (key2 == 0)
                    key2 = key3;
                /* ---- shift Function keys allowed ---- */
                if ((k<4 && key2) && (key2 < 59 || key2 > 68))
                    buzz();
                else if (!(k && (key2 || multibits(k))))
                    buzz();
                else
                    printf("\r%s                                    ",
                        showkey(key2, k));
            }
        }
        /* -------- wait for the key release -------- */
        while ((kbval & 0x80) == 0)
            ;
    }
    if (key1 == 57 && k == 0)
        *keymsk = *keyscan = 0;
    if (key1 == 28 && k != 0)    {
        *keymsk = k;
        *keyscan = key2;
    }
    clearkb();
    *bp = svbios;

    endkey();
    kbval = 0;
}

/* ----- test a nybl for multiple bits ---- */
static int multibits(int k)
{
    int ct = 0, i;

    for (i = 1; i < 16; i *= 2)
        if (k & i)
            ct++;
    return (ct > 1);
}

/* ----- show the hot key ------ */
char *showkey(unsigned keyscan, unsigned keymsk)
{
    *hotky = '\0';
    if (keymsk & 8)
        strcat(hotky, "Alt-");
    if (keymsk & 4)
        strcat(hotky, "Ctrl-");
    if (keymsk & 2)
        strcat(hotky, "LShift-");
    if (keymsk & 1)
        strcat(hotky, "RShift-");
    if (keyscan)
        strcat(hotky, scodes[keyscan-1]);
    else
        *(hotky + strlen(hotky) - 1) = '\0';
    return hotky;
}

/* ----- clear the BIOS keyboard read-ahead buffer ----- */
static void clearkb(void)
{
     int far *nextoff = MK_FP(0x40,0x1a);
     int far *nexton  = MK_FP(0x40,0x1c);
     *nextoff = *nexton;
 }

 static void buzz(void)
 {
     sound(100);
     delay(250);
     nosound();
 }
```

End Listings

Listing One *(Text begins on page 139.)*

```
UNIT ArrObj;

{ A simple string array object   }
{ with display and menu methods  }
{ to demonstrate one need for    }
{ multiple inheritance.  Needs   }
{ Turbo Pascal 5.5.              }
{ By Jeff Duntemann              }
{ Presented in DDJ for July 1990 }


INTERFACE

USES Crt,BoxStuff;

CONST
  LowBounds = 0;
  HighBounds = 14;


TYPE
  String40 = STRING[40];

  StringArray = OBJECT
                  Data  : ARRAY[LowBounds..HighBounds] OF String40;
                  Index : Integer;
                  CONSTRUCTOR Init;
                  PROCEDURE SetIndex(NewIndex : Integer);
                  FUNCTION  GetIndex : Integer;
                  PROCEDURE SetCurrent(NewString : String40);
                  FUNCTION  GetCurrent : String40;
                  PROCEDURE Display(X,Y : Integer);
                  FUNCTION  Choose(X,Y : Integer) : String40;
                END;


IMPLEMENTATION

VAR
  I       : Integer;
  MyArray : StringArray;


PROCEDURE ShowReverse(X,Y : Integer; Target : String40);

VAR
  Save : Word;

BEGIN
  Save := TextAttr;
  TextColor(Black);
  TextBackground(LightGray);
  GotoXY(X,Y); Write(Target);
  TextAttr := Save;
END;


PROCEDURE ShowNormal(X,Y : Integer; Target : String40);

VAR
  Save : Word;

BEGIN
  Save := TextAttr;
  TextColor(LightGray);
  TextBackground(Black);
  GotoXY(X,Y); Write(Target);
  TextAttr := Save;
END;


PROCEDURE Uhuh;

VAR I : Integer;

BEGIN
  FOR I := 1 TO 2 DO
    BEGIN
      Sound(50);
      Delay(100);
      NoSound;
      Delay(50);
    END;
END;



{----------------------------}
{ Method definitions follow:  }
{----------------------------}

CONSTRUCTOR StringArray.Init;

VAR
  I : Integer;

BEGIN
  { Clears the strings in the array to null: }
  FOR I := LowBounds TO HighBounds DO
    BEGIN
      SetIndex(I);
      SetCurrent('');
    END;
END;


PROCEDURE StringArray.SetIndex(NewIndex : Integer);

BEGIN
```

(Listing continued on page 150.)

Listing One *(Listing continued, text begins on page 139.)*

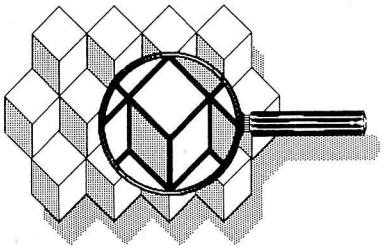
```
  Index := NewIndex;
END;


FUNCTION  StringArray.GetIndex : Integer;

BEGIN
  GetIndex := Index;
END;


PROCEDURE StringArray.SetCurrent(NewString : String40);

BEGIN
  Data[Index] := NewString;
END;


FUNCTION  StringArray.GetCurrent : String40;

BEGIN
  GetCurrent := Data[Index];
END;


PROCEDURE StringArray.Display(X,Y : Integer);

VAR
  I : Integer;

BEGIN
  MakeBox(X,Y,42,HighBounds+3,GrafChars);  { Show a box }
  FOR I := LowBounds TO HighBounds DO
    BEGIN                                  { and display the strings }
      GotoXY(X+1,Y+1+I); Write(Data[I]);   {  inside it. }
    END;
END;



{ This is a VERY simple bounce-bar menuing method for    }
{ the string array object.  No scrolling is done; if you }
{ define more elements that will fit on the screen, you  }
{ must allow for the user's scrolling the list up and    }
{ down within the window defined by the line character   }
{ box.                                                   }

FUNCTION StringArray.Choose(X,Y : Integer) : String40;

VAR
  I : Integer;
  Done,EscPressed : Boolean;
  Ch : Char;

BEGIN
  Display(X,Y);
  { Highlight the element at Index: }
  ShowReverse(X+1,Y+1+Index,Data[Index]);
  I := Index; Done := False; EscPressed := False;
  { Get user input until Enter or Esc is pressed: }
  REPEAT
    REPEAT {Null} UNTIL KeyPressed;
    Ch := ReadKey;
    CASE Ch OF
      #0  : BEGIN  { 0 means an extended key code follows: }
              Ch := ReadKey;
              CASE Ch OF
                #72 : BEGIN  { Up arrow }
                        ShowNormal(X+1,Y+1+I,Data[I]);
                        IF I = LowBounds THEN UhUh  { Up Arrow }
                          ELSE Dec(I);
                        ShowReverse(X+1,Y+1+I,Data[I]);
                      END;
                #80 : BEGIN  { Down arrow }
                        ShowNormal(X+1,Y+1+I,Data[I]);
                        IF I = HighBounds THEN UhUh { Down Arrow }
                          ELSE Inc(I);
                        ShowReverse(X+1,Y+1+I,Data[I]);
                      END;
              END; { CASE }
            END;
      #13 : Done := True;  { Enter pressed; we're done }
      #27 : BEGIN
              EscPressed := True;  { Don't change anything }
              Done := True;
            END;
    END; { CASE }
  UNTIL Done;
  IF EscPressed THEN  { Put things back as they were: }
    BEGIN
      ShowNormal(X+1,Y+1+I,Data[I]);
      ShowReverse(X+1,Y+1+Index,Data[Index]);
    END
    ELSE
    BEGIN
      Index := I;              { I becomes the new index }
      Choose := Data[Index];   {  and we return the string at Index }
    END
END;

END.
```

End Listing One

Listing Two

```
PROGRAM MenuArray;

{ Demonstrates a string array object   }
{ with its own built-in menuing method }
{ By Jeff Duntemann      }
{ From DDJ for July 1990 }


USES Crt,ArrObj;

VAR
  I        : Integer;
  MyArray  : StringArray;
  MyString : STRING;


FUNCTION PullRandomString(DoLength : Integer) : STRING;

VAR
  I : Integer;
  TempString : STRING;

FUNCTION Pull(Low,High : Integer) : Integer;

VAR
  I : Integer;

BEGIN
  REPEAT                    { Keep requesting random integers until }
    I := Random(High + 1); { one falls between Low and High }
  UNTIL I >= Low;
  Pull := I
END;

BEGIN
  FOR I := 1 TO DoLength DO
    TempString[I] := Chr(Pull(32,125));
  TempString[0] := Chr(DoLength);
  PullRandomString := TempString;
END;


BEGIN
  ClrScr;
  MyArray.Init;            { Set up the object }
  FOR I := 0 TO 14 DO      { Fill the array with random strings }
    BEGIN
      MyArray.SetIndex(I);
      MyArray.SetCurrent(PullRandomString(40));
    END;
  MyArray.SetIndex(7);     { Point the index to element 7 }
  MyString := MyArray.Choose(5,2);   { Invoke menu method }
  GotoXY(6,22); Writeln(MyString);   { Display the chosen string }
  Readln;
END.
```

End Listings



Version 2.1 of the Zortech C++ Development System for MS-DOS and OS/2 should be available this month from Zortech. Version 2.1 includes improvements in compilation speed, optimization, and generated code quality.

Zortech claims that with this version they have overcome the DOS memory limitations by combining their Virtual Code Management (VCM) technology with the DOS extender technology of Rational Systems. VCM allows MS-DOS applications to contain up to 4 Mbytes of code and still run in real mode. The VCM system requires changes only to assembly language code.

Applications with large and complex class hierarchies can now be compiled on PCs with 80286/386 or 486 processors. You can also use the compiler to develop applications using the Rational Systems DOS extender. They've used the same technology in the debugger — it relocates itself into extended memory, allowing large programs to be debugged. A virtual 8086 debugger for 386-based systems is provided, and requires little conventional memory.

Zortech is also developing versions of the C++ Developers Edition for Xenix and Unix 386. The 32-bit compiler will be compatible with the Phar Lap 386/DOS Extender.

Contact Zortech for pricing information on C++ 2.1. Reader service no. 34.
Zortech Inc.
4C Gill St.
Woburn, MA 01801
617-646-6703
800-848-8408

Graphics programmers who manage megabyte-hungry images can turn to the ECOMP software module from **EFI (Electronics for Imaging)** for help. With ECOMP, details and colors in different image areas undergo different levels of compression by using algorithms that analyze spatial frequencies; the compression algorithm selectively preserves coefficients and approximates or drops others. Images are then reconstructed from these coefficients, but changes from the originals are those the eye is least likely to detect. The level of filtering is controlled by user-specified quality factors.

In high-quality electronic publishing, typical images can undergo a compression ratio of 20:1 without noticeable difference in image quality. The software is supplied in C source code and is operational on Mac IIs with Symantec's Think C, on IBM PC/ATs and compatibles with Borland's Turbo C, and on Sun Microsystems Sun/4 with the Sun C compiler. Compression speed averages 5 to 20 Kbytes of image data per second, depending on computer architecture. The package includes programs in TIFF file format, which are portable to other file formats such as TARGA and PICT2. EFI provides the product as C source code, so you can integrate it with other programs and as part of a developed system. Contact EFI for licensing information. Reader service no. 20.
EFI
950 Elm Ave.
San Bruno, CA 94066
415-742-3400

New from **Borland** comes Turbo C++, a full implementation of AT&T's C++ 2.0. Turbo C++ includes a standalone compiler and a compiler within Borland's new user interface, Programmer's Platform. Turbo C++ includes mouse support, multiple overlapping windows, a multifile editor, an intelligent project manager, an integrated debugger, and transfer capability for accessing other programs.

Also included is VROOMM, the virtual run-time object-oriented memory manager that lets you overlay your code without complexity; acommand-line compiler, linker, and tools; online hypertext help with copy and paste examples; and libraries such as heap-checking functions and a set of complex and BCD math functions. All this comes in the Turbo C++ package for $199. If you order Turbo C++ Professional you also get Turbo Debugger 2.0, Turbo Assembler 2.0, and Turbo Profiler 1.0 for $299. Reader service no. 22.
Borland International
1800 Green Hills Rd.
P.O. Box 660001
Scotts Valley, CA 95066
408-439-1619

If you want 8514/A compatibility, you can save lots of money by purchasing the RIXAI video driver from **RIX Softworks**. The RIXAI video driver is a software-based 8514/A emulator that is compatible with any extended VGA board. This product accounts not only for the 8514/A video board and compatible display, but also for the 8514/A driver, the HDILOAD Adapter Interface (AI).

Because hardware specs were unavailable from IBM, RIX had to program through the HDILOAD driver and create an AI for extended VGA boards that would appear as a hardware implementation of the 8514/A Adapter Interface.

RIXAI will benefit application developers and board manufacturers by reducing the software development efforts required for extended VGA boards. If the user's display is 8514/A-compatible, all software that supports the 8514/A adapter interface will get immediate access to all of the VGA boards supported by RIXAI. RIXAI does not support scissors or plane-enable functions due to the high memory and performance overhead associated with these functions. In 16-color mode, the emulator does not support color mixing.

The RIXAI video driver has been successfully tested with all the major "non-Windows" applications that offer 8514/A support. All standard 8514/A display resolutions (640 × 480 and 1024 × 768 with 256 colors, and 1024 × 768 in 16 colors) are provided by RIXAI. The RIXAI video driver is available free to ClubRIX extended support members (714-476-0728). Otherwise the cost is $100. Doc Livingston told *DDJ* that "this is more than $1,000 less than the cost of an 8514/A video board." Reader service no. 40.
RIX Softworks
18552 MacArthur Blvd., Ste. 200
Irvine, CA 927715
714-476-8266

Other Mac news: **Pixar** has announced MacRenderMan, the Mac version of PhotoRealistic RenderMan. Pixar intends for RenderMan to be the standard for 3-D scene description. MacRenderMan accepts data in the RIB (RenderMan interface byte stream) file format from Mac-based 3-D design applications. Applications spool RIB files into a MacRenderMan folder and MultiFinder processes them in the background, similar to the printing process. Rendered images are then output to a color display or a file. MacRenderMan can output images in PICT, EPS, and TIFF file formats; these images can then be used by Mac-based 3-D design and multimedia applications.

DDJ spoke with Sean McKenna of Paracomp, which produces 3-D modeling packages for presentations, product design, and graphic arts. About MacRenderMan, he said "our company is real excited. This is a completely

(continued on page 158)

(continued from page 153)
new category for the Mac — nothing like this was available before. It lets our objects and models look more realistic than ever. Whereas before we needed a large machine to produce images of this quality, we can now do them on the desktop, though they still take a long time."

Mac application developers can either embed MacRenderMan in their software or offer a shrink-wrapped version with the product. MacRenderMan includes a RenderMonitor for managing jobs in a background queue, an application for viewing images on a color Mac and for managing the display and conversion of TIFF or PICT files, picture-making software for generating rendered 3-D images, ShaderApp for compiling and managing shaders written in the RenderMan shading language, and a sample library of RIB files, shaders, and texture maps. You'll need a Mac SE 30. II, IIx, or IIci running System 6.0.3 or later with MultiFinder and 32-bit QuickDraw, 8- or 24-bit color display, 4 Mbytes of disk storage, and at least 4 Mbytes of memory. Reader service no. 28.

Pixar
3240 Kerner Blvd.
San Rafael, CA 94901
415-258-8100

Grasp, the animation language and paint program for IBM PCs and compatibles, is now multimedia. **Paul Mace Software**'s new version offers CD sound control and special animation effects. Other enhancements include direct memory management, file handling capabilities, math and string operators, and differential animation techniques.

Steve Grumette, of Artificial Intelligence Research Group, is a big Grasp fan. He does computer effects for the film industry, and told *DDJ* that he uses Grasp to generate effects for the television show *Alien Nation.* A programmer who is also a graphics artist, Grumette likes the "easy-to-code animation techniques and the ability to do real-time animation. You can create a picture with a paint program and manipulate it with the Grasp language. One of the improvements in this new package is the addition of variables in the program." These give Grasp the computational power of languages such as C and Pascal.

Grasp supports the following systems: IBM compatible XT/AT/PS2 with hard drive (640K RAM suggested); CGA, EGA, VGA, Hercules, and other display cards up to 1024 × 768; PC Paintbrush, Gem IMG, CompuServe GIF, Basic BSAVE image formats, Video capture/overlay boards, printers, CD-ROM drives, and more. It retails for $199. Reader service no. 29.

Paul Mace Software
400 Williamson Way
Ashland, OR 97520
503-488-2322

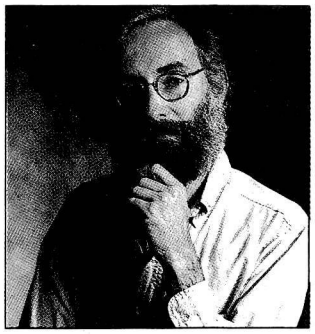
Version 5 of the TIFF Development Library for PCs is now available from **Image Software**. This new version includes full support for reading and writing images in the TIFF Level 5 format and also reads images stored in Level 4 format. Level 5 support includes several compression schemes, such as LZW, Packbits, CCITT Group 3, bilevel, and gray scale, and supports all TIFF classes. The Library is compatible with Microsoft Windows or DOS applications, and was developed in Microsoft C.

The Development Library contains a dump utility for analyzing TIFF files and for debugging applications that import images from other programs. Version 5 comes with two sample programs for demonstration. It sells for $95. Reader service no. 31.

Image Software
P.O. Box 1634
Danville, CA 94526
415-838-4244

DDJ

Caution! Man At Work



This month's "Flames" is a rare watch at the artist at work, showing how I write this column every month.

Mike and Nancy's machine here. Start talking.

Hi Cuz, this is Corbett. Listen, if you're still planning to flame about the misuse of the word "product," don't do it. I looked into it and "product" without a preceding article is not an error. In fact it makes a useful distinction. "A product" is something produced for sale, but "product" without the "a" is stuff produced for sale. With "a product," you conceive of the thing before you write the marketing plan, and with "product," it's the other way around. Take a look at David Gerrold's piece in the *New York Times Book Review*; I think it's the April 29th issue. He sort of says that a science fiction novel is "a product," while a fantasy novel is "product." Although we're hearing the usage more all the time in the software industry, it probably just represents wishful thinking on the part of sales types who wish they were peddling something as ephemeral as Andre Norton novels. Later, man.

Beep.

Mike and Nancy's machine here. Start talking.

Michael. This is Stan. I heard that you were going to be writing about assigning credit for the discovery of the Mandelbrot set, which would seem to be the same species of question as who's buried in Grant's Tomb or who writes "Swaine's Flames." But if, in discussing Mandelbrots and fractals and chaos theory, you get around to assigning credit for the invention of chaos, you might consider Mr. Pournelle. He's brought more than his share of chaos into our industry, in a Manor of speaking, and I understand he's in the Soviet Union now, so you can defer repercussions, should he be unamused. Jerry Pournelle in the Soviet Union is itself an amusing concept, don't you think? I have it on good authority that he gave as his reason for going that he wanted to go there before the only place a communist could be found was on an American university campus. Must go; I have some deviltry to advocate. Take care.

Beep.

Mike and Nancy's machine here. Start talking.

Mike, it's Oliver. Dig this: Now that the sordid saga of est and the founding of Computerland is being serialized in *PC Computing* and a biography of Philippe Kahn is in the works, it's time to do the blockbuster Silicon Valley movie. It's a great subject, better than Wall Street, better than Vietnam. I see it as a pretentious big-budget epic with a lot of box-office stars and dialog straight out of the soaps, and I've got some thoughts about plotlines and casting that I'm hoping you can run by your readers. Like, picture Tom Cruise and Rick Moranis as two kids who build a computer in a garage, right? They become insanely rich, but it goes to their heads. The Moranis character tries to produce a rock concert, but he has a nerd's taste in music. The Cruise character throws tantrums and bullies everybody but has incredible charisma and personal magnetism. Eventually he's kicked out of the company but comes back by inventing a sexy black-box computer, which will be played by a Cromemco Z2D. In a separate plotline, what's his name, you know, the nerdy kid who plays Paul Pfeiffer in *The Wonder Years*, is a software magnate who's always in these heavy meetings with three-piece suits from IBM. But at the same time he's battling it out with John Candy, who plays a rogue, try-anything-once software developer selling compilers on street corners. He plays sax at Moranis' rock concert, so there's a tie-in there. Then we've got Bill Murray drifting in and out of things as a disk jockey who gets into software and becomes the spreadsheet king, but he runs around in a Hawaiian shirt all the time, and toward the end of the film he's giving peptalks on software design. Suck in those guts, guys, we're Software Designers. You get the idea. He represents the creative spirit. Oops. Got another call. Let's do lunch.

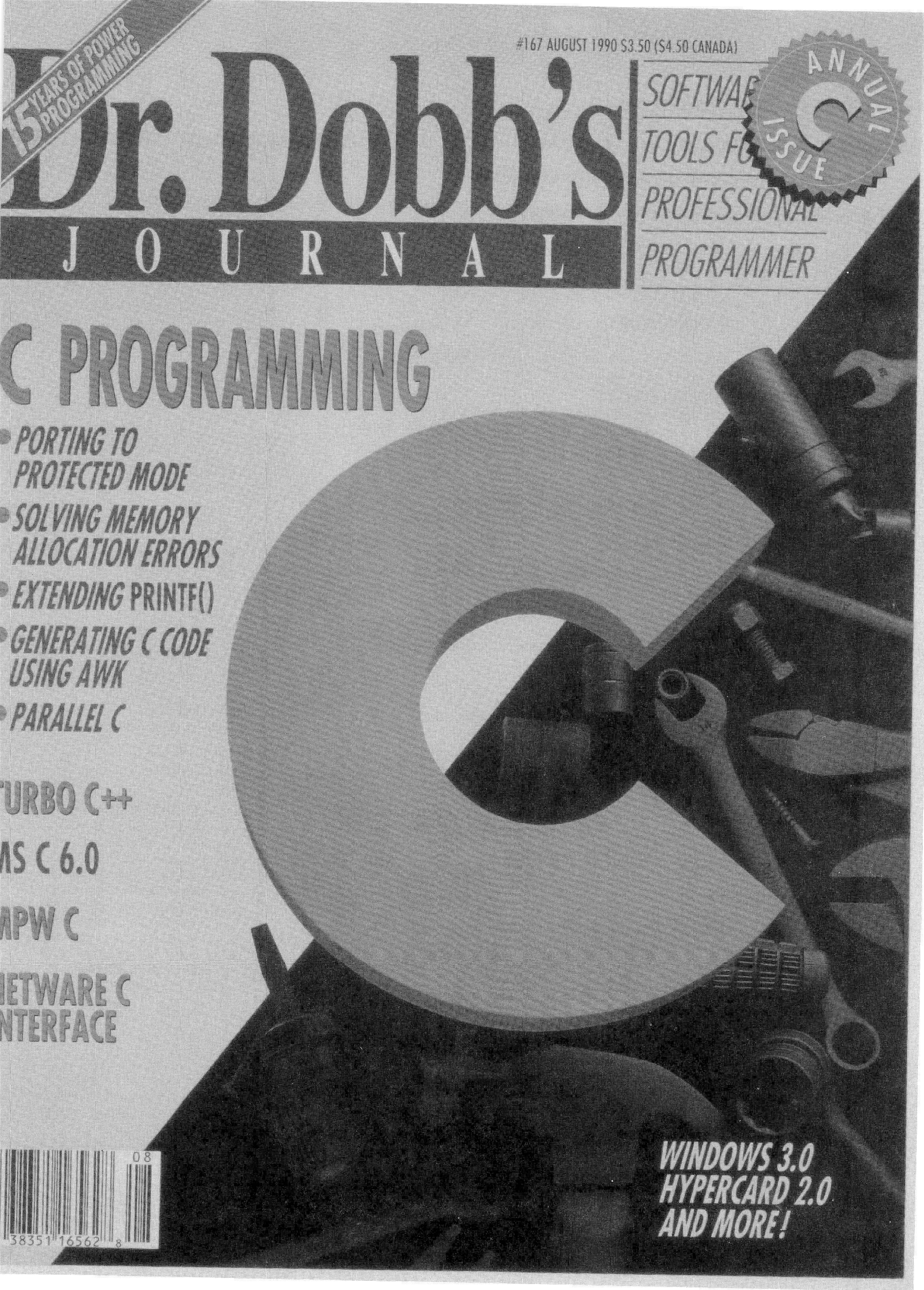
Beep.

Mike and Nancy's machine here. Start talking.

Michael. This is Stan again. It occurred to me after my last call that there is in fact some question regarding who writes "Swaine's Flames." You should know that someone is circulating the rumor that you compile the column by transcribing answering machine messages. I suspect your cousin Corbett of starting that one. A more disturbing one is that your cousin Corbett has himself been writing the column for the past year, and that it's entirely fictional. Since I have made one or two appearances in "Swaine's Flames," would that make me a fictional character? That would be a shallow existence, to live only in prose. The only consolation would be to have the last word.

Beep.

Michael Swaine


15 YEARS OF POWER PROGRAMMING
#167 AUGUST 1990 $3.50 ($4.50 CANADA)
Dr. Dobb's
JOURNAL
SOFTWA
TOOLS FO
PROFESSIONAL
PROGRAMMER
ANNUAL C ISSUE
C PROGRAMMING
PORTING TO PROTECTED MODE
SOLVING MEMORY ALLOCATION ERRORS
EXTENDING PRINTF()
GENERATING C CODE USING AWK
PARALLEL C
TURBO C++
MS C 6.0
MPW C
NETWARE C INTERFACE
WINDOWS 3.0
HYPERCARD 2.0
AND MORE!
08
38351 16562 8

CONTENTS

AUGUST 1990
VOLUME 15, ISSUE 8

FEATURES

PORTING C PROGRAMS TO 80386 PROTECTED MODE 16
by William F. Dudley, Jr.
Bill shares his experiences and suggestions for porting large C programs to 80386 protected mode.

ENCAPSULATING C MEMORY ALLOCATION 24
by Jim Schimandle
Encapsulating memory routines gives you control over memory allocation. Jim presents a memory shell that does this.

AWK AS A C CODE GENERATOR 36
by Wahbah Baldwin
When you need to perform tasks such as converting data from one format to another — and generate C code in the process — AWK may be the tool for you.

IMPLEMENTING BICUBIC SPLINES 48
by Raymond G. Lauzzana and Denise E.M. Penrose
Raymond and Denise mix and match C and Lisp to generate a spline function that uses the Macintosh Toolbox to draw a smooth curve.

EXTENDING *printf( )* 60
by Jim Mischel
Here's an easy-to-use variation on *printf( )* that utilizes variable arguments so you can output formatted numbers.

PARALLEL EXTENSIONS TO C 70
by Graham K. Ellis
Ken discusses concurrency functions for parallel C compilers and describes how to build transputer-based parallel systems.

DEBUGGING MEMORY ALLOCATION ERRORS 80
by Lawrence D. Spencer
Larry tackles C memory allocation using functions that serve as a bookkeeping layer between programs.

HANDLING OS/2 ERROR CODES 134
by Nico Mak
Here's how to squeeze more information out of OS/2's cryptic error messages.

EXAMINING ROOM

OPTIMIZING WITH MICROSOFT C 6.0 84
by Scott Robert Ladd
Scott examines the most recent version of Microsoft's C compiler, focusing on global optimization, while Bruce Schatzman provides additional comments on based pointers.

PROGRAMMER'S WORKBENCH

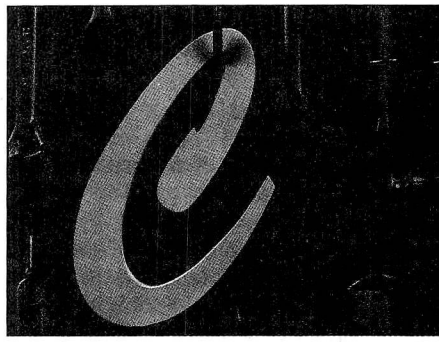
COLLECTIONS IN TURBO C++ 94
by Bruce Eckel
Collections are classes that hold a number of object types. Bruce uses Borland's Turbo C++ to create this class and analyzes the C++ 2.0 specification.

COLUMNS

PROGRAMMING PARADIGMS 137
by Michael Swaine
What do Windows 3.0, HyperCard 2.0, and Norman Mailer 1.0 have in common? Michael ponders this and other questions.

C PROGRAMMING 149
by Al Stevens
Al mulls over C's past and future before exploring the NetWare programming environment.

STRUCTURED PROGRAMMING 161
by Jeff Duntemann
Jeff creates a Turbo Pascal object to solve a recent data-entry problem.



DEPARTMENTS

EDITORIAL 6
by Jonathan Erickson

LETTERS 8
by you

SWAINE'S FLAMES 184
by Michael Swaine

PROGRAMMER'S SERVICES

ADVERTISER INDEX 184
where to go for more information on products

OF INTEREST 185
compiled *by Janna Custer*

PROGRAMMER'S MARKETPLACE 186
classified ads

SOURCE CODE AVAILABILITY

As a service to our readers, all source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Drive, Redwood City, CA 94063, or call 800-356-2002 (inside Calif.) or 800-533-4372 (outside Calif.). Specify issue number and disk format. Code is also available through the *DDJ* listing service (415-364-8315) and through the DDJ Forum on CompuServe (type GO DDJ).

NEXT ISSUE

September's Structured Language issue puts Pascal, Modula-2, Fortran, and other languages to work solving a variety of programming problems.

Our 1991 Editorial Calendar

The future is now, or so it seems as we begin planning next year's editorial line up. Thanks in part to the thousands of you who responded to our call for 1991 article topic suggestions, we've settled on the following monthly themes for the upcoming year.

January	Software Design
February	Data Compression
March	Assembly Language Programming
April	Biocomputing
May	Programming for Coprocessors
June	Structured Languages
July	Graphics Programming
August	C Programming
September	Little Languages, Big Engines
October	Object-Oriented Programming
November	Operating Systems
December	User Interfaces

Before going any further, I'll quickly throw in my standard caveat. These aren't the only topics we'll be discussing; in most issues of *DDJ*, we try to devote at least half the magazine to non-theme topics. Among those "other" subjects we'll be covering are embedded systems programming, data structures, 32-bit programming, communications, scientific and engineering programming, windowing systems, software engineering, and just about any efficient implementation of an algorithm.

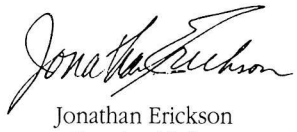
Keep in mind that we prefer advance notice on all articles, particularly if you want to match your article with a specific monthly theme. However, we'll run a good technical article on any aspect of programming in any month. And remember, the more source code (in any language), the better.

If you have an article in mind but want some guidance on how to prepare the proposal and article, give us a call or drop us a letter and we'll get you going.

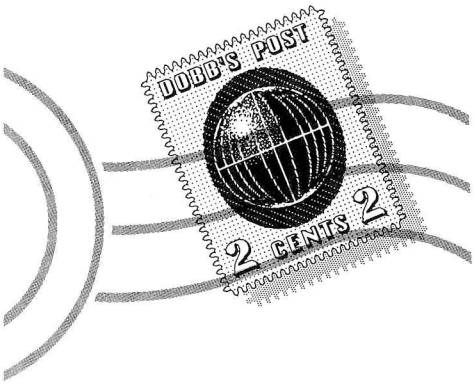
A special thanks to Terry Vaughn of West Covina, California, the first reader to make a donation to the Kent Porter Scholarship Fund. In his note, Terry says " . . . if everyone does the same, it'll add up." We've received numerous requests for applications and we're looking forward to sharing Terry's goodwill with others.

Those of you who've been using the *DDJ* listing service should note that the system has moved from New Hampshire, to California, with the phone number changing to 415-364-8315. David Betz will continue development work from New Hampshire while Keith Lyon will be the programmer on this coast.

And at least one software company took to heart my mention a few months back of Dr. Dobbs, the race horse we adopted as our official mascot. A recent letter from the company in question was addressed to "Mr. Jon Erickson, Ed." but, mailmerge programs being what they are, the note began "Dear Mr. Ed,". No, it didn't hurt my feelings (I've been called a lot worse). I just whinnied, stamped my foot on the floor three times, and made my way out to pasture.



Jonathan Erickson
editor-in-chief



Dining Philosophers Discussion

Dear *DDJ*,

In "Programming Paradigms," *DDJ* issue #164, entitled "Complex Systems, Fractals, and Chaos," Michael Swaine briefly discusses a problem known as the "Dining Philosophers" problem. This problem was first stated and solved by E. W. Dijkstra in "Cooperating Sequential Processes" (Technical Report EWD-123. Eindhoven, The Netherlands: Technological University, 1965). Since Dijkstra's original paper, this problem has been considered a classic process synchronization problem, not because it has practical importance, but because it is an example for a large class of concurrency control problems.

The problem may be stated as follows, derived from the description in *Operating Systems Concepts* by James L. Peterson and Abraham Silberschatz, (Reading, Massachusetts: Addison-Wesley, 1985). Five philosophers spend their entire lives thinking and eating. These philosophers share a circular table with five place settings. The table is set with five plates of rice, and five chopsticks. When a philosopher thinks, he does not interact with the other four philosophers. On occasion, a philosopher gets hungry and tries to pick up the chopsticks to his left and right, in either order, so that he can eat a plate of rice. A philosopher may only take one chopstick at a time, and, obviously, cannot take a chopstick that is in use by one of his neighbors. When a hungry philosopher has both of his chopsticks, he eats without releasing his chopsticks until he finishes. When finished eating, a philosopher puts both of his chopsticks back on the table and starts thinking again.

In reference to this problem, Mr. Swaine makes the following statement on page 123,

> The Dining Philosophers problem, discussed here and more fully in David Harel's book, *Algorithmics* (Addison-Wesley, 1987), is a classic case of deadlock that cannot be resolved without introducing an element of randomness. Any strictly deterministic solution to the Dining Philosophers problem is guaranteed to fail.

Apparently, either Mr. Swaine or Mr. Harel has not done his homework thoroughly enough, as this assertion is at odds with the information to be found in the relevant literature. Since the column is not specific as to whether Mr. Harel's book makes this same assertion or not, I cannot tell whose homework did not get done.

This assertion is also at odds with the concepts taught at the university level. I received a Bachelor of Science in Information and Computer Science from the Georgia Institute of Technology in 1987. This curriculum included a senior-level operating systems course, in which each student must solve the dining philosophers problem in a manner that is both deterministic and *provably* free of both deadlocks and starvation. Further, each student must provide a proof of correctness with the solution. My own class implemented the solution using Logitech Modula-2 and an instructor-provided module allowing preemptive multitasking of multiple instances of a single procedure. The Peterson and Silberschatz book was the text for this course when I attended it, and was the genesis of our programming assignment.

Peterson and Silberschatz provide three solutions which are guaranteed to be free of deadlocks:

1. Allow at most four of the philosophers to be seated at the table simultaneously.
2. Force each philosopher to take his chopsticks, eat, and release his chopsticks, within a critical section, thus allowing only a single philosopher to eat at any instant.
3. Use asymmetry: Odd numbered philosophers take one chopstick first (e.g., the one on the right) while even numbered ones take the other (e.g., the left).

All of these solutions can be proven to be free of deadlocks. The correctness proofs for these three solutions are not provided in the text, because, unfortunately, none of them meets the additional criterion that starvation must be avoided. The author indicates that a deadlock- and starvation-free solution is possible but leaves the development, implementation, and proof of correctness of the solution as an exercise for the reader.

The discussion of this problem in *Operating Systems: Design and Implementation* by Andrew S. Tanenbaum (Englewood Cliffs, New Jersey: Prentice-Hall, 1987) casts it in a slightly different light, but differing from the characterization of Peterson and Silberschatz in details only: In Tanenbaum the philosophers are trying to eat very slippery spaghetti which requires two forks. Mr. Tanenbaum goes into more depth with his discussion of the possible solutions. The following quotation is from page 77 of Tanenbaum's book (the figures mentioned are included at the end of this letter, following the references). [*Editor's Note*: See Examples 1 and 2.]

> Now you might think, "If the philosophers would just wait a random [amount of] time instead of the same [amount of] time after failing to acquire the right-hand fork [chopstick], the chance that everything would continue in lock step for even an hour is very small." Of course this is true, but in some applications one would prefer a solution that always works and cannot fail due to an unlikely series of random numbers. (Think about safety control in a nuclear power plant.)
>
> One improvement to Fig. 2 – 19 [see Example 1] that has no deadlock and no starvation is to protect the five statements following the call to think by a binary [mutual exclusion] semaphore. . . .
>
> From a theoretical viewpoint, this solution is adequate. From a practical one, it has a performance bug [inefficiency]: Only one philosopher can be eating at any instant. With five forks [chopsticks] available we should be able to allow two philosophers to eat at the same time.
>
> The solution presented in Fig. 2 – 20 [see Example 2] is correct and also allows the maximum parallelism for an arbitrary number of philosophers.

The net result of this discussion is that Mr. Swaine's assertion is false. The problem *can* be solved in a deterministic manner. Additionally, it *must* be solved in a deterministic manner in order for the solution to be *provably* correct. I know this is true both from the literature and from the experience of having developed, implemented, and proven correct a deterministic solution devoid of deadlocks and starvation.

I find it quite disturbing that Mr. Swaine would assert that a sequencing problem of this sort can be solved with randomness. Nondeterminism can definitely solve the problem, since a nondeterministic solution can be found to

(continued on page 12)

(continued from page 12)
be further optimized:

```
cmp al, 10
sbb al, 69h
das
```

which is one byte shorter and four cycles faster on an 8088 (although with the 8088's prefetch queue, maybe I should say "one byte faster," not one byte shorter . . .). Thanks again for a great magazine.

Tim Lopez
San Jose, California

OOP ASM Recommendations

Dear *DDJ*,

I was pleased to see the article in the March 1990 *DDJ* about OOP assembly language. I was surprised, though, to see that messaging was not implemented by the author. I used the same techniques as Mr. Hyde as far as structures go, but in addition I added messaging as a simple jump table. Each object handled its own messages, eliminating the need for the caller to know the procedure name of the desired method.

Greg Messer
Houston, Texas

Setting the Mandelbrot Straight

Dear *DDJ*

I enjoyed Michael Swaine's "Programming Paradigms" column on fractals in the May 1990 *DDJ*. For the record, however, I would like to mention that by no means is Benoit Mandelbrot the "discoverer" of fractals. Indeed, Mandelbrot deserves credit for studying them, popularizing them, and coining the term "fractal." But von Koch had discovered the snowflake curve by 1904, and Hausdorff defined and studied fractional-dimensional sets in 1919.

Also, to get slightly technical, self-similarity is *not* a requirement for a set to be fractal, at least not in Mandelbrot's own definition; the set must only have fractional Hausdorff dimension.

These facts are all extracted from Mandelbrot's *Fractal Geometry of Nature* (Freeman, 1982), albeit with difficulty.

Daniel Asimov
Berkeley, California

Parental Guidance

Dear *DDJ*,

I read with interest Jeff Duntemann's column "Grinding the Speckled Axe" (*DDJ*, May 1990) in which he talks about the difficulty of drawing the line between parent and child. He gave two object hierarchies for accepting user input:

1. Parent *Field*, child objects *StringField*, *IntegerField*, and *DateField*;
2. Parent *StringField*, child objects *IntegerField* and *DateField*. The second hierarchy is more code efficient, but precludes character-by-character entry validation.

What interests me is that this situation is not specific to OOP, but actually occurs in any language that supports functions or procedures. The first hierarchy is roughly equivalent to writing functions *GetStringField( )*, *GetIntegerField( )*, and *GetDateField( )*. (Admittedly, the common code would have to be copied manually rather than neatly encapsulated as in the *Field* object.) The second hierarchy is equivalent to writing a function *GetStringField( )*, and then writing functions *GetIntegerField( )* and *GetDateField* that call *GetStringField( )* to get raw data, and then validate it appropriately. The first solution duplicates a lot of code, but it allows each character to be validated as the user types. The second solution is code-efficient, but the user's errors aren't detected until <Enter> is pressed. Which is better? I don't know, either. There's still no firm answer after all these years of procedural languages, and I suspect there never will be.

Steve Corwin
Shelton, Connecticut

TopSpeed Replies

Dear *DDJ*,

This is in response to the April 1990 "C Programming" column by Al Stevens. I was gratified to see that Mr. Stevens likes TopSpeed C, and does not hesitate to recommend it to others. However, Mr. Stevens does raise several points about TopSpeed C to which I'd like to respond.

TopSpeed C *does* allow an application to call *INTERRUPT* functions. However, it must be done through the library's *_CHAIN_INTR ( )* function.

While TopSpeed C does not provide the *_FLAGS* pseudovariable, it is fairly easy to implement such a function. TopSpeed C allows the programmer to control many aspects of how functions are called. This function declaration allows the CPU's flags to be read, as in Example 3. Similarly, this function declaration allows the CPU's flags to be set, as in Example 4. In these two examples, compiler pragmas are used to:

1. Save the state of the pragmas;
2. Specify that functions will preserve a specific list of registers;
3. Specify that functions are to be expanded in-line; and
4. Restore the saved state of the pragmas. The functions themselves are specified as a sequence of machine code values. (I have given these two definitions in a form which allows them to be passed directly into a header file for inclusion into a program.)

As you can see, TopSpeed C provides the programmer with a considerable amount of power. Since no C compiler could possibly provide intrinsic functions for everything that every programmer would want to do, we designed TopSpeed C to allow the programmer to define functions which are called with the same efficiency as those which are intrinsic to the compiler.

Mr. Stevens's comments on the documentation are correct. Fortunately, we only made a small print run of this set of manuals. We will make corrections to the manuals for inclusion in the next print run, and will make the corrected manuals available to customers with an older set.

Mr. Stevens has provided us with useful recommendations on how to improve the WATCH utility. While a fully configurable utility (as Mr. Stevens wishes WATCH was) could be marketed as a product all by itself, we will continue to provide WATCH as a utility within the TopSpeed products and are working to expand its capabilities.

Don Dumitru
JPI
Mountain View, California

```
#pragma save
#pragma call(reg_saved=>(bx,cx,dx,si,di,ds,es,st1,st2), inline=>on)
static int _flags(void) =          {0x9c /* pushf */,
                                    0x58 /* pop ax */};
#pragma restore
/*  usage:  i = _flags();  */
```

Example 3

```
#pragma save
#pragma call(reg_saved=>(ax,bx,cx,dx,si,di,ds,es,st1,st2),
inline=>on)
static void _set_flags(int i) = {0x50 /* push ax */,
                                 0x9d /* popf */};
#pragma restore
/*  usage: _set_flags(i); */
```

Example 4

Porting C Programs to 80386 Protected Mode

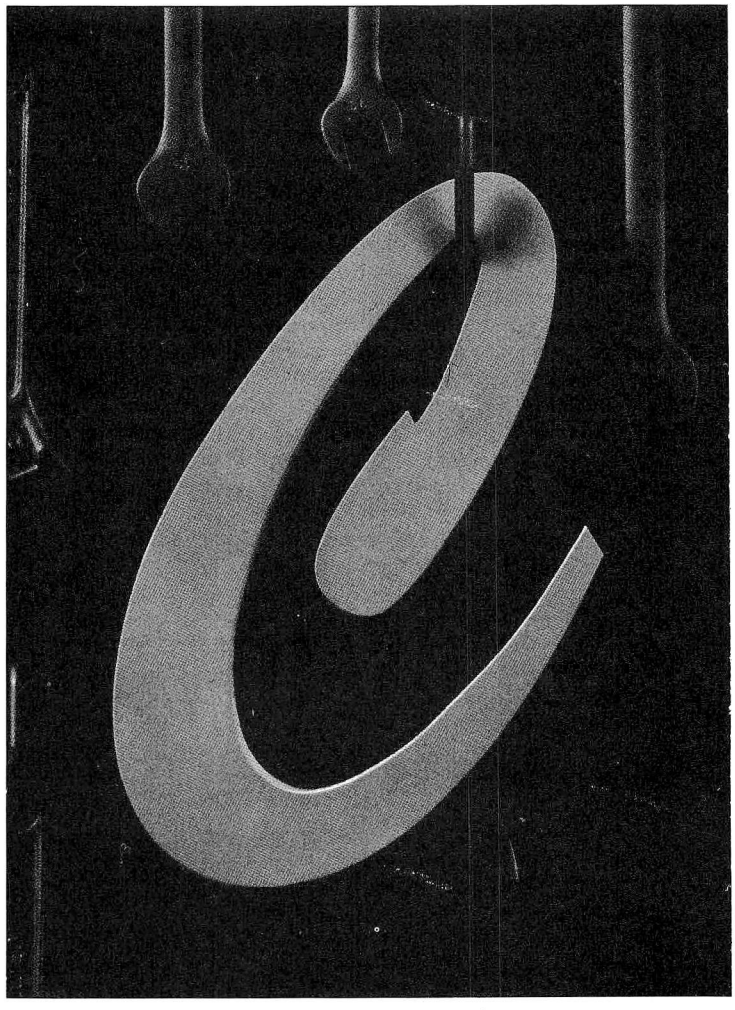
When you need more speed and greater capacity

William F. Dudley, Jr.

As one of the programmers at a developer of CAD software for printed circuit-board design, I was assigned the job of porting our CAD programs from Microsoft C 4.0 to an 80386 protected-mode compiler. After making a quick study of the three available compilers that support the 80386 — NDP from Microway, High-C 386 from MetaWare, and Watcom 7.0/386 from Watcom — we chose the Watcom compiler. This article describes the problems and solutions we encountered in the process of porting 75,000 lines of C.

We had two programs to port, the autorouter and the graphical editor. The autorouter, called, oddly enough, ROUTER, was the first candidate for porting for two reasons: It was written by a very small team so it was a lot "cleaner," and because it was not interactive, the speed of its graphics output was, for the time being, not important. The second program to port was the graphical editor, Draftsman-EE (called DM). In this case, video display graphics speed is very important, so we expected to spend more development work here.

We had two goals when we began the port: Achieving greater speed and increased capacity. First of all, we figured that accessing data in one large (multi-megabyte) array would be faster than all the rinky-dink calculations we were doing to calculate which EMS page the data was in, and then get it in the real-mode version of our product. Secondly, we felt that the real-mode product was limited by the size of an unsigned *int*. The protected mode (32 bit) advantage is obvious.



Bill is director of engineering for Design Computation Inc., a vendor of CAD software for printed circuit design. Bill has a masters in electrical engineering from Cornell University. When not programming for profit or fun, he can be found riding his Norton Commando (on nice days) or working on that or one of his other bikes (on less nice days). He can be reached at RD5, Box 239, Jackson, N.J. 08527.

In an attempt to quantify the speed improvement, we simulated it by making a small model version of ROUTER that had no paging to EMS of data elements. It ran about 25 percent faster than the normal real-mode ROUTER, so that was what we expected. (See the "Summary" at the end of this article for the surprising result.)

Moving to an ANSI Compiler

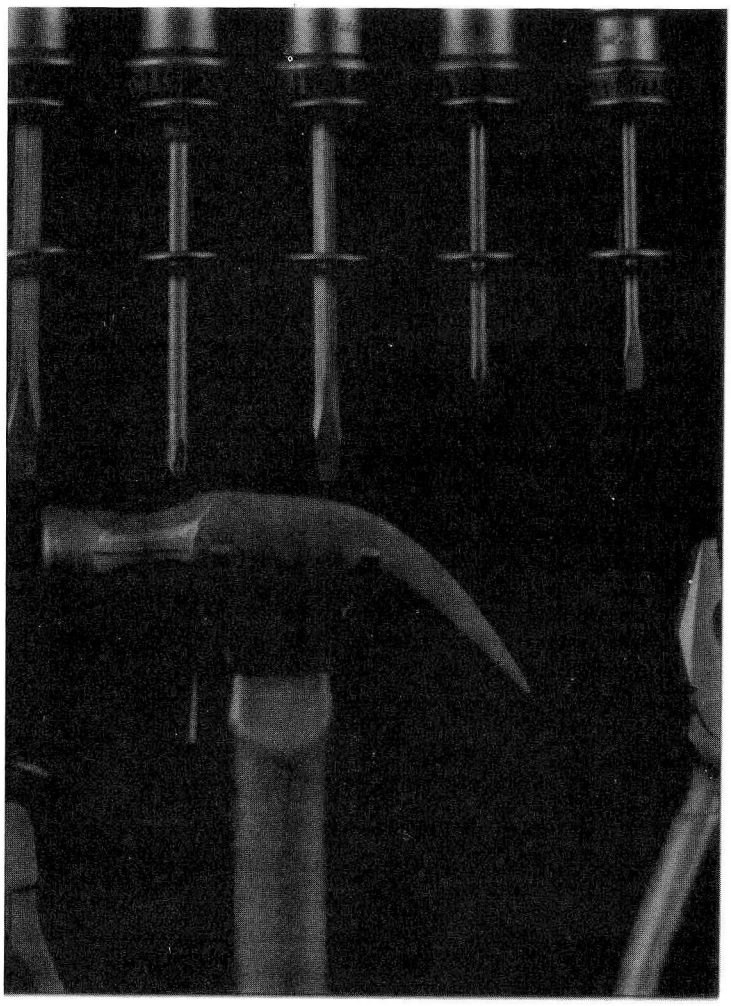
This is probably old news to everybody now, but I thought I'd bring up the ANSI compiler issue and attempt to explain it again for those of you who are out of touch. The big deal in moving to an ANSI compiler is that the function headers change from:

```
integer_fn(i, f, c) int i; float f; char c; { . . . }
```

to:

```
int integer_fn(int i, float f, char c) { . . . }
```

This would be harmless enough in itself, except for the following gotcha: *Pre-ANSI compilers always promoted floats to doubles in function argument lists.* ANSI compilers pass floats as floats if you so declare. And here is what caused me a few interesting hours: *The ANSI compiler will pass floats to the function when it is called, but the function will expect doubles when it runs, unless you use the new style function header.* The moral of this story is that you may continue to use old function headers in your code, except for functions that expect doubles or floats as arguments, in which case you *must* use the new style headers.



Increased Memory Usage

An obvious side effect of moving to protected-mode operation is that integers are now 32 bits unless declared "short." The non-obvious result is that structures containing *int*s grow, memory usage increases, and any code that hardcoded the size of the struct breaks horribly. I know, we shouldn't have hardcoded them, but we did, and we had a reasonably good excuse.

Now I have a hairy preprocessor macro that computes the size of a "data element," which is really the biggest of any of several different structs. Some judicious juggling of the struct contents also reduced the maximum struct size to save some memory.

The Programming Challenges

Our CAD programs required some special services that are not available from DOS. The first one was the ability to talk to a "security device," more popularly known as a "parallel port dongle." This really presented two problems: How to link a real-mode module to a protected-mode program, and how to pass the character string data from the protected-mode application to the real-mode dongle code.

The second service — not available from DOS — is device independent video graphics. We solved this problem in our real-mode programs by supplying a family of video driver TSR (terminate and stay resident) programs. Now we needed to figure out how to talk to our video drivers from protected mode.

Talking to Real-Mode Assembly Code Modules

The manufacturer of our dongle, Rainbow Technologies, supplies an assembly language module that talks to the dongle, and can be linked to a Microsoft C program. This enables you to interrogate the dongle from C without getting your hands dirty. A protected-mode program, however, needs some tricks in order to link it with the supplied real-mode module.

The code in Listing One (page 104) is a stripped-down version of the assembly code dongle module. Notice that I chose to pass the arguments directly in the registers instead of on the stack. This is because I have to use Phar Lap's facility for calling real-mode code, which allows me to use the registers easily.

The other noteworthy item is that the data buffer is in the real-mode code segment. This was done because it was simpler than figuring out how to have a real-mode data segment and getting it to link in the proper order.

The code in Listing Two (page 104) is the C module that talks to the assembly module of Listing One. It has two functions: Initializing variables to point to the various parts of the real-mode code and doing the actual call to interrogate the dongle.

Three locations in the real-mode code are directly accessed from protected mode as shown in Example 1. The linker allows us to know the address of anything in the real-mode section, so that gives us the protected-mode version of those addresses. The Phar Lap DOS extender has a function for finding out the equivalent real-mode address of any protected-mode address, so the subroutine *pr2real( )* returns the real-mode address at the start of the real-mode code as a (32 bit) integer. Subroutine *real_setup( )* uses this result to initialize *real_addr*, *real_seg*, and *real_off*.

The actual dongle communication occurs in the subroutine *COMPUTE( )*, which uses Phar Lap function *0x2510* to call the real-mode routine *QUERY_FAR*. Before the call is made, the *blk_mv_pr( )* routine copies the string to be tested to the buffer in the real-mode code segment. *Blk_mv_pr( )* and its sinister twin, *blk_mv_rp( )* are shown in Listing Three (page 105).

Dr. Dobb's JOURNAL
SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

PUBLISHER *Peter Hutchinson*

EDITORIAL
EDITOR-IN-CHIEF *Jonathan Erickson*
MANAGING EDITOR *Monica E. Berg*
TECHNICAL EDITORS *Michael Floyd, Ray Valdes*
ASSOCIATE MANAGING EDITOR *Janna Custer*
CONTRIBUTING EDITORS *Al Stevens, Jeff Duntemann, Martin Tracy, David Betz, Tom Genereaux, Andrew Schulman*
COPY EDITORS *Rhoda Simmons, Pamela Dillehay, Nan Fornal*
EDITOR-AT-LARGE *Michael Swaine*

ART/PRODUCTION
ART/PRODUCTION DIRECTOR *Larry L. Clay*
ART DIRECTOR *Michael Hollister*
PRODUCTION SUPERVISOR *Amy Shulman Lesovoy*
TYPOGRAPHERS *Teresa Raines, Margaret Anderson, Charlene Carpentier*
COVER PHOTOGRAPHER *Michael Carr*

CIRCULATION
DIRECTOR OF CIRCULATION *Maureen Kaminski*
CIRCULATION MANAGER *Randy Robertson*
CIRCULATION PLANNING MANAGER *Manny Sawit*
DIRECT MARKETING MANAGER *Andrea Weingart*
NEWSSTAND MANAGER *Sarah Forsman*
DIRECT MARKETING COORDINATOR *Susan Bauman*
PROMOTION COORDINATOR *Philip Tsang*
FULFILLMENT COORDINATOR *Anne Jean*

ADMINISTRATION
VICE PRESIDENT OF FINANCE *Kate Deschamps*
CONTROLLER *Mary Collopy*
CREDIT MANAGER *Betty Arsene*
ACCOUNTING SUPERVISOR *Renate Kernke*
ACCOUNTS RECEIVABLE *Wendy Ho*
ACCOUNTS PAYABLE *LuAnn Rocklewitz*

MARKETING/ADVERTISING
ASSOCIATE PUBLISHER *Karla Spormann*
ADVERTISING COORDINATOR *Laura Stack Pullen*
MARKETING ASSISTANT *Sara Noah Ruddy*
ACCOUNT MANAGERS *see page 184*

M&T PUBLISHING INC.
CHAIRMAN OF THE BOARD *Otmar Weber*
DIRECTOR *C. F. von Quadt*
PRESIDENT *Laird Foshay*
VICE PRESIDENT OF PUBLISHING *William P. Howard*

DR. DOBB'S JOURNAL (USPS 307690) is published monthly, except semimonthly in December, by M&T Publishing, Inc., 501 Galveston Dr., Redwood City, CA 94063; 415-366-3600. Second-class postage paid at Redwood City and at additional entry points.

ARTICLE SUBMISSIONS: Send manuscripts and disk (with article, listings, and letter to the editor) to the associate managing editor 415-366-3600.

***DDJ* ON COMPUSERVE:** Type GO DDJ.

SUBSCRIPTION: $29.97 for 1 year; $56.97 for 2 years. Foreign orders must be prepaid, including the additional postage (air or surface) in U.S. funds drawn on a U.S. bank. Add $13 per year for surface mail; add $36 for airmail to Canada and Mexico; or $26 for airlift to all other countries.

POSTMASTER: Send address changes to *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. **ISSN 1044-789X**

CUSTOMER SERVICE: For subscription questions, call toll-free 800-456-1215. For subscription orders or change of address call toll-free 800-456-1215 (U.S. and Canada) or write *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. For book/software orders call 800-533-4372 (in California 800-356-2002).

FOREIGN NEWSSTAND DISTRIBUTOR: Worldwide Media Service Inc., 115 E. 23rd St., New York, New York 10010; 212-420-0588 FAX 212-420-1265.

Entire contents copyright ©1990 by M&T Publishing, Inc., unless otherwise noted on specific articles. All rights reserved.

The Audit Bureau

Passing Data Between Real- and Protected-Mode Code Sections

The second challenge is that of passing chunks of data between real- and protected-mode code sections. When you request a DOS service, for example writing to the disk, the Phar Lap DOS extender handles this by copying the disk data from your buffer in protected mode to another buffer in real mode, and then running DOS. This happens transparently so that you never know how messy it is.

An obvious side effect of moving to protected-mode operation is that integers are now 32 bits unless declared "short"

If you are doing something abnormal, however, you quickly find out how messy it can get. Our video drivers need to exchange data with the application in two cases: Telling the application what the palette of the video board is (so the user can change color assignments) and passing large blocks of pixels to and from the application for screen save/restore operations.

The answer to this is to use the Phar Lap facility for reserving a block of "real-mode" memory (memory guaranteed accessible by a real-mode program). Listing Four (page 105) shows the graphics video module in the application. Subroutine *setvmode( )* initializes the video driver as well as establishing the location (in both real- and protected-address spaces) of the 9-Kbyte buffer we use for communication with the driver. Subroutine *rstr_vbuf( )* restores the screen image from a previously saved file. (Subroutine *save_vbuf( )* is not shown but does exactly the reverse.)

Talking to a Real-Mode Graphics Driver from Protected Mode

The third challenge arose when we wanted to communicate with our video drivers. The actual communication of drawing requests seemed easy enough, because our driver is accessible through its own (software) interrupt. Easy it was, but deadly slow.

This wasn't really a surprise, since our real-mode product had long ago given up using software interrupts for communication with the video driver. In the real-mode product, the video driver returns at initialization time the segment and offset of its main entry point. The application makes a pointer to a function out of this, and calls the driver directly just as if it was part of the application code.

The problem was much worse in the protected-mode product, however. This was due to the overhead of switching the machine from protected to real mode and back again

```
extern char test_string;
       /* string buffer in real mode module */
extern char end_real,
       /* end of real mode code */
extern char QUERY_FAR;
       /* actually a function, but we just want
                                          address */
```

Example 1: *Three locations in the real-mode code are directly accessed from protected mode.*

(continued from page 18)
for every vector sent to the video driver. The solution was obvious — another programming challenge! We would stuff the driver commands in a list, and when the list filled up, throw it over the wall to the real-mode code, which would then call the video driver for each item in the list. This would reduce the overhead of the real protected-mode change by a factor equal to the length of the list.

In Listing Four, notice the two lines:

```
prot.vidfn.addr[1] = r.x.si;
prot.vidfn.addr[0] = r.x.di;
```

These stuff the address that the driver returns (the address of its entry point) into *struct prot* for later use by the real-mode list interpreter. Listing Five (page 108), the include file, defines this structure (see Example 2). This structure actually occurs twice: Once on the protected side and again on the real-mode side. Whenever the protected-mode code runs the draw list interpreter, it first copies the protected struct over to the real-mode version, and then uses Phar Lap function *0x2510* to call the real mode draw list interpreter.

```
struct {
  union {
   int (* p)();
     short int addr[2];
     /* [0]segment and [1]offset of driver entry point */
    } vidfn ;
   short int lp;
   short int list[LLEN][6];
} prot ;
```

Example 2: *Protected-mode structure*

Listing Six (page 108) is the assembly language module for the protected-mode subroutine *kdidraw( )*, which stuffs draw commands on the draw list. If the draw list fills up, it automatically calls the protected-mode subroutine *pdinterp( )* to empty it.

The assembly language modules were generated by coding the problem in C and then optimizing the assembler output of the compiler

Listing Seven (page 109) shows the C code of the protected-mode side of the draw list interpreter. Subroutine *pdintinit( )* sets up the registers structs and subroutine *pdinterp( )* calls the real-mode interpreter. (This is very similar to Listing Four, which calls the real-mode dongle module.)

Finally, Listing Eight (page 109) is the assembly code of the real-mode subroutine that "interprets" the draw list. It is just a tight loop that pushes six *int*s from the list onto the stack and calls the subroutine pointed to by the segment and offset at the beginning of the RAM locations labelled *_real*. It does this repeatedly until the first integer in the group of 6 is 0. This is the end-of-list marker. The *_real* location is the real-mode copy of the prot struct.

The assembly language modules were generated by coding the problem in C and then optimizing the assembler output of the compiler. The reason was simply to maximize the speed because the overhead of the video calls was slowing us down.

Before I embarked upon the coding of the protected-mode version of the draw list interpreter, I built up a version in Borland's Turbo C. This enabled me to test the performance as well as debug bits of it with a source debugger before committing to the protected version. The C code from the Turbo C version was not in itself useful for the protected-mode version, but the experience of building and debugging it was worth the time.

Summary

How did all this turn out? Boy, the speed increase really surprised us. The protected ROUTER runs twice as fast as the real-mode product (if the video is turned off). The protected graphics editor, Draftsman-EE, also shows a factor of two improvement for compute bound processes. Video speed appears the same as in the real-mode product, which is probably due to the vectors being generated in real-mode code. The next job is to make a protected-mode video driver to fix that bottleneck. As for increased capacity, no customer has yet tried a job that is greater than 65,535 elements. (The real-mode product will do a 300 IC board, which is pretty big.)

DDJ

(Listings begin on page 104.)

Vote for your favorite feature/article.
Circle Reader Service **No. 1.**

Encapsulating C Memory Allocation

Detect memory allocation errors automatically

Jim Schimandle

Originally the program was small, less than 20,000 lines, and command-line driven. But after two years, four different programmers, and the addition of a lot of functionality — a nifty graphical user interface, multiple printer support, foreign language menus, and a kitchen sink — the program grew to 150,000 lines. Then it started crashing.

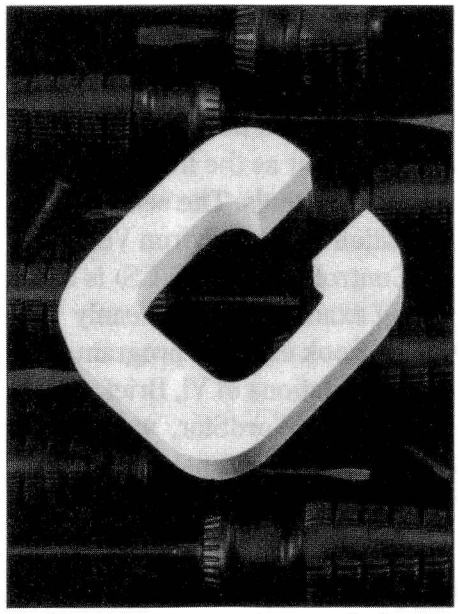
The timing of crashes was erratic. Sometimes the program would run for weeks without crashing, at other times it would crash in two days. After a few weeks of debugging, I finally realized that the program was running out of dynamic memory. I had a classic case of memory leakage.

What is Memory Leakage?

When C programmers allocate memory using *malloc( )*, they must be sure to *free( )* the memory when done using it. If the memory is not freed, it "leaks" out of the usable memory pool. Such memory cannot be reused.

Consider the program in Listing One, page 110. This program allocates a list of items until all memory is used up. Then the list of items is freed. If the program is designed properly, it should never exit the *while* loop in *main( )*.

Jim owns Primary Syncretics, which specializes in the design of hardware and software for embedded, real-time systems. His 10 years of experience runs the gamut from panel switch bootstraps to SPARC data acquisition systems. Jim can be reached at 408-988-3818.



When this program is compiled and run, however, it eventually runs out of memory and exits the *while* loop.

The problem is found in the *junk_close( )* function. The function correctly frees the *JUNK* structure, but fails to free the string pointed to by *junk_name*. (See lines 68-83 and 86-89 of Listing One.)

This Will Never Happen to Me

I hear you cry foul. "This is nothing more than an example of poor programming," you're saying. "This would never happen to a real programmer." Besides, if another programmer looked at the code, it would be obvious where the problem lies.

First, if you look at the revision history in lines 2-8 of Listing One, the genesis of the error is more understandable. The name tag was added by a different programmer after the first release of the module. We can assume the second programmer had a problem, added a quick fix, and then forgot to look into the other effects the change might have on the program.

Secondly, this is a pared-down version of the original 150,000-line application. Finding this error in such a large application is a daunting task. You could look at this code for weeks and probably not notice the lack of a single *free( )* in one function.

Finally, in the original application, the calls to the *open* and *close* functions were data dependent. Thus the programming error showed up only under certain data conditions.

Examining Assumptions

Whenever I am faced with a failure in a system, I search for the basic assumptions that lead to the failure. For the standard library memory allocation calls, the underlying assumption that leads to memory leaks is that the caller is responsible for error checking.

This assumption is a direct result of the Unix history of C. The entire Unix philosophy assumes that programmers must handle all errors, at all times, with no exceptions. There is no "big brother" on Unix. You're on your own.

Well, a peon like myself needs a slightly more bullet-proof interface.

(continued from page 24)
What I really need is embedded code that can inform me when a problem occurs. Then, when I make a mistake, the mistake will be automatically caught during development.

By encapsulating the memory allocation functions, I can provide a layer of protection between my code and the system library. Such a memory shell intercepts the system library calls and performs basic bookkeeping and error checking.

Designing with a Purpose

Initially, I was interested only in detecting a memory leak. However, the actual memory shell realized multiple design goals:

- Memory leaks are detected.
- Manipulation of an invalid memory block is reported.
- The location within the client code which generates an error is reported.
- There is minimal execution overhead.
- The shell does not require massive changes to existing source code.

Junk Revisited

Before we look at the internals of an actual memory shell, let's look at how the shell can be used. Listing Two, page 110, contains a modified version of the junk program in Listing One, with only two changes. The first is the inclusion of the mshell.h interface header (lines 11-13 of Listing Two). The second is the calling of the *Mem_Used( )* function in *main( )*, lines 29-44 of Listing Two.

When the program executes, it produces the output shown in Example 1. Because the program reaches the *printf( )* within the *if* statement (line 38, Listing Two), the value returned by *Mem_Used( )* must have been nonzero. This indicates that not all allocated memory was freed. *Mem _Display( )* produces the other information in the output. For each block not freed, the amount of memory and the file/line number where the allocation was made is displayed.

You can see that the most common error was the failure to free a 26-byte block. Line 83 in Listing Two shows

```
*** Memory list not empty ***
Index   Size   File(Line) - total
                      size 6238
0       6004   junk1.c(80)
1         26   junk1.c(83)
2         26   junk1.c(83)
3         26   junk1.c(83)
4         26   junk1.c(83)
5         26   junk1.c(83)
6         26   junk1.c(83)
7         26   junk1.c(83)
8         26   junk1.c(83)
9         26   junk1.c(83)
```

Example 1: *Output from junk1 shows that there are really two sources for the memory leaks. The first comes from the* malloc( ) *of the* JUNK *structure at line 80. The second comes from the* strdup( ) *of the name at line 83.*

that this is the string allocated by *strdup( )*. The fix is equally obvious. Simply add a *free( )* of *junk_name* to *junk_close( )*. •

However, another error is also revealed. Somehow, a *JUNK* structure allocated on line 80 has not been freed. This error is harder to determine and requires a reexamination of the logic in *junk_open( )*. If the allocation of the *JUNK* structure on line 80 succeeds but the allocation of the string on line 83 fails, *junk_open( )* fails to free the *JUNK* structure before returning a *NULL*.

The second error would be hard to detect by inspection and is also dependent on the amount of memory allocated by other modules. This type of error can be further masked if the allocation statements in *junk_open( )* are written using a typical C coding style that attempts to run as many assignment statements as possible into a single *if* conditional, as in Example 2. (Why any rational human being would want to do this is beyond me.) This example demonstrates the utility of a memory shell in tracking down some pernicious bugs.

Redirecting the Standard Library

How can I simply add an *include* file and suddenly have error checking of the memory interface? The names of the memory routines have not been changed. Yet somehow, I am able to get C to call my routines and not the standard library.

Pointer Alignment

The standard library routines *malloc( )*, *realloc( )*, and *calloc( )* always return a pointer of suitable alignment for an object of any type. Alignment refers to a power of 2 address boundary that a processor "naturally" operates on. This natural boundary is normally related to the bus width or specific processor architecture limitations.

A SPARC processor requires that a *long* be aligned on a 4-byte boundary. This means that the address for a *long* must have zeros in the least significant 2 bits. If an attempt is made to load a *long* from a misaligned address, an alignment fault is detected and the program crashes.

An Intel 80386, however, has no requirements on alignment. You are allowed to load any data type from any address. However, if you attempt to load a *long* from an address with ones in the least significant 2 bits, the processor must perform two memory loads to get the data. This overhead is unacceptable in most applications. So, although the processor does not enforce alignment, you would be a fool not to take advantage of the bus bandwidth.

Processor	Minimum Size	Suggested Size
8088	1	1
80286	1	2
80386	1	4
80386sx	1	2
68000	2	2
68020	1	4
SPARC	8	16
VAX	1	16

Table 1: *The alignment sizes for various processors depend upon system bus width and architecture limitations. Failure to observe the minimum size will cause a processor fault. The suggested size is based on knowledge of the bus width and cache line sizes. Larger alignment sizes can cause larger alignment gaps.*

To meet the alignment requirement, the size of a memory block header must be rounded up to the nearest alignment boundary. The bytes in the resulting alignment gap cannot be used by either the memory shell or the client code. The alignment gap lies between the memory block header and the client data region. (Refer to Figure 1.)

The memory shell uses *ALIGN_SIZE* to define the alignment byte boundary required for the largest object that must be aligned. *ALIGN_SIZE* is used by the *RESERVE_SIZE* macro, which rounds up the size of the memory header to the next *ALIGN_SIZE* boundary. The resulting *RESERVE_SIZE* is used by the pointer conversion macros. Suggested values for *ALIGN_SIZE* for various processors can be found in Table 1. —J.S.

(continued from page 26)

To see how this is accomplished, you must understand that C macros are expanded by a preprocessor before the compilation phase. If a macro is defined with the same name as a standard library function, the macro definition is expanded before compilation. The compiler actually sees the macro expansions, not the standard library calls. Thus, a *malloc( )* call in client code is converted to a *mem_alloc( )* call by the macro *malloc* in mshell.h (see Listing Three, page 110). To the client, *mem_alloc( )* must provide the same service as the standard library function it replaces. This technique is called "redirection."

For example, in Listing Two the source code call *malloc(sizeof(JUNK))* on line 80 is actually a request for the preprocessor to expand the macro *malloc* defined in mshell.h. This is expanded by the preprocessor to *mem_alloc((size of(JUNK)),"junk1.c",103)*. Since the C compilation phase sees only the token *mem_alloc( )*, the call is correctly redirected to the memory shell.

In order to use redirection, you must be able to recompile the source code that calls the library. With this caveat, redirection can be implemented for any system or user library. You can even perform redirection without changing any source code by using compiler command-line macro definitions.

I have used redirection to debug code that interfaces with system library routines such as *fopen( )* and *fclose( )*. I often use redirection to check modules written by other programmers.

Other Interface Features

If *MEM_WHERE* is defined, the redirection macros expand into function calls that contain filename and line information. With this information, error messages can pinpoint the client code that allocated the memory block. The *__FILE__* and *__LINE__* macros are not available with all compilers, so turn on *MEM_WHERE* only if compiler support is available.

If *MEM_LIST* is defined, the building of an internal list of memory blocks is enabled. This option is used internally in mshell.c. The resulting list is used for dumping the entire allocated memory list when an error occurs.

The header file also defines two functions not found in the standard library: *Mem_Used( )* and *Mem_Display( )*. These functions are used to check memory usage and display the current memory list, respectively.

The redirection of the standard library occurs only if *__MSHELL__* is not defined. The mshell.c module defines *__MSHELL__*, which allows mshell.c to call the standard library.

Memory Block Header

In order to provide added error checking, the memory shell must store additional data for each allocated memory block. Since we have no idea how many memory blocks will be allocated, a static array is insufficient. We could perform a separate *malloc( )* to allocate storage for a control structure every time a new memory block is created. A more effi-

```
if (((jnew = (JUNK *) malloc(sizeof(JUNK))) == NULL) ||
    ((jnew->junk_name = strdup(name)) == NULL))
        {
        return NULL ;
        }
```

Example 2: *Obtuse coding for the allocation of memory in* junk_open( ) *can hide the possibility of a memory leak when the first allocation succeeds and the second allocation fails.*

(continued from page 28)
cient method, however, is to allocate a memory block large enough to store both the control structure and the data region. Only the memory shell functions are allowed to access the data in the control structure. Only the memory shell clients are allowed to access the data region.

In the memory shell, the control structure is called the "memory block header." This header is placed before (at lower addresses in relation to) the data region. See Figure 1.

The standard library functions use pointers to the data region. Since this does not correspond to the base of the memory block allocated by the memory shell, we have to convert to and from memory block header pointers and client data region pointers. This is done by using pointer arithmetic and casts. The macros *CLIENT_2_HDR* and *HDR_2_CLIENT* handle these pointer conversions (See lines 74-75, Listing Four, page 111.)

Header Fields

The actual fields within the header vary depending on which compilation options are used (see lines 45-57 of Listing Four). At a minimum, tag and size values are required.

The tag is a unique value that identifies this memory block as valid. The tag field is set to the unique value when the header is first initialized. All memory shell routines that manipulate the header first check for the validity of the tag before performing any operations. While there is a finite possibility that the tag value will be valid for an invalid block, the odds are very small.

The size is the number of bytes in the client data region. This value is needed to keep track of the total number of bytes currently allocated by the memory shell.

If *MEM_LIST* is defined, next and previous links are added to the header. These pointers are used to build a two-way linked list of memory blocks. This list is used for displaying the memory blocks in use.

If *MEM_WHERE* is defined, a pointer to a filename and a line number are added to the header. These fields are filled in by the last memory shell function to manipulate the block. Note that the filename string is not copied locally. Thus, the filename must be either in static or global storage. This rule is followed by all compilers that I have used that support the *__FILE__* feature.

Shell Implementation

The function *mem_alloc( )* receives calls intended for *malloc( )*. First, *mem_alloc( )* allocates a memory block large enough for both the memory block

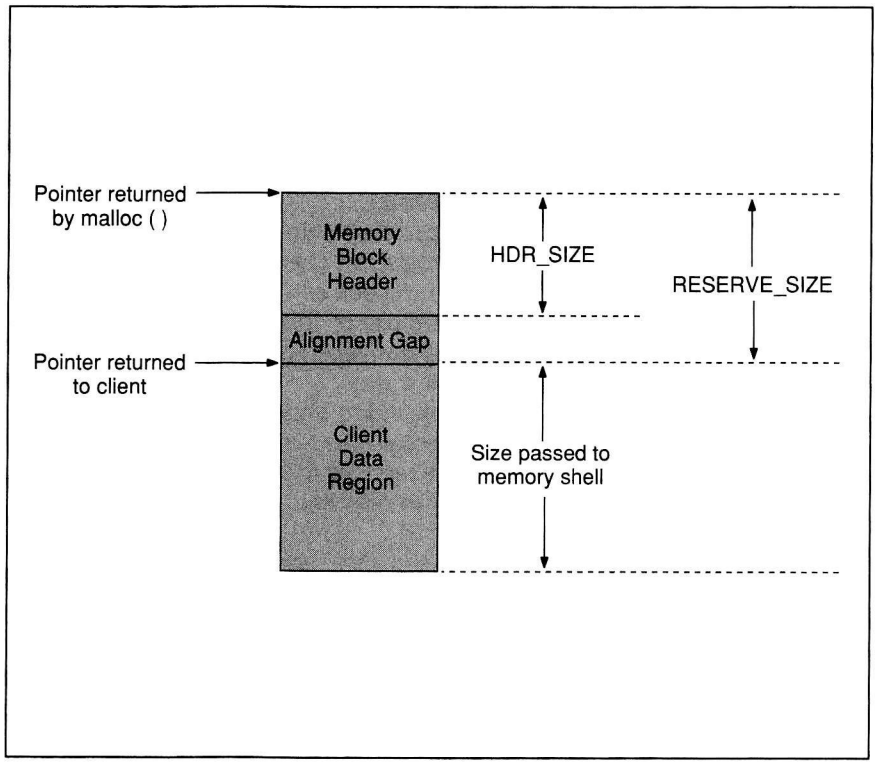


Figure 1: *A memory block header is added to every memory block allocated. This structure is used to store information needed to implement error checking. The alignment gap is an artifact of data type alignment restrictions enforced by various processors.*

(continued from page 30)
header and the client data region using *malloc( )*. If the allocation fails, a *NULL* is returned to the client. If the allocation succeeds, the memory header is initialized. If *MEM_LIST* is enabled, the memory block is added to the memory block list. Finally, the *HDR_2_CLIENT* macro is used to convert a memory block header pointer to a client data region pointer. The resulting pointer is returned to the client.

Note that this routine does not check for errors. It simply stores enough information to allow for error checking in *mem_realloc( )* and *mem_free( )*.

The function *mem_free( )* receives calls intended for *free( )*. First, *mem_free( )* converts the client data region pointer to a memory header pointer using the *CLIENT_2_HDR* macro. If the tag in the header is invalid, an error is reported. The tag is then complemented to force the tag to be invalid. This allows the detection of a duplicate *free( )* of the same memory block. If *MEM_LIST* is enabled, the memory block is removed from the memory block list. Finally, the standard library *free( )* function is used to return the memory block to the system.

The function *mem_realloc( )* receives calls intended for *realloc( )*. The logic is a combination of that found in *mem_free( )* and *mem_alloc( )*.

The function *mem_strdup( )* receives calls intended for *strdup( )*. In *mem_strdup( )*, a call is made to *mem_alloc( )* to obtain the needed memory. If the call fails, a *NULL* is returned. Otherwise, the string is copied to the client data region and the string pointer is returned to the client.

Bad memory tags are reported using *mem_tag_err( )*. This function should be modified to suit the needs of your particular application. It currently displays an error message, dumps the memory list, and crashes the program. Not graceful, but acceptable for development.

The detection of a memory leak requires a check immediately before your program exits. If your termination code correctly frees all allocated memory, *Mem_Used( )* should return zero. If not, you can use *Mem_Display( )* to look at the memory blocks that are still allocated. (See lines 36 – 41 in Listing Two.)

The functions *mem_list_add( )* and *mem_list_delete( )* manipulate the two-way linked list of memory blocks. The list manipulation is standard, so I won't dwell on it.

I have not included *calloc( )* in the memory shell because I never use the function. If you wish to use *calloc( )*, the implementation is straightforward. Calculate the size needed, call *mem*

_alloc( ), and then clear the data region with *memset( )*.

Performance and Space Issues

The memory shell adds overhead both in execution speed and in the amount of memory allocated. In the wide range of applications I have coded, the overhead is not noticeable and has never deterred me from using the shell. If the overhead causes problems in your application, you can conditionally turn the protection features on for development and then disable the more time-and space-consuming features for production.

This shell, in various incarnations, has worked on many MS-DOS machines, Sun3s, Sun4s, and VAXs

I tend not to disable any error checking for production. Instead, I conditionally compile the code to dump enough information to allow me to debug the problem when the customer finds that inevitable bug. When I can debug the problem quickly, customer goodwill far outweighs the performance or speed penalties.

Portability

This shell, in various incarnations, has worked on many MS-DOS machines, Sun3s, Sun4s, and VAXs. The portability problems I've encountered are:

- If your compiler lacks *__FILE__* and *__LINE__* support, undefine *MEM_WHERE*.
- If your compiler does not support ANSI prototypes or *#if defined( )*, you will have to make changes to the code for backward compatibility.
- If your processor enforces alignment for data types, the value of *ALIGN_SIZE* may need to be adjusted. See the accompanying text box entitled "Pointer Alignment" for more information.
- If you need to use system calls directly to take advantage of extra functionality, add a secondary memory shell. This shell provides calls analogous to the standard library calls, but will take advantage of system-specific features. I recently used such a

shell on a port from MS-DOS to OS/286. The shell allocated each memory block as a separate LDT entry. This allowed hardware checking for many pointer-related errors. I was amazed at the number of pointer errors that were detected in shipping "bug-free" code.

Variations on a Theme

Since this shell was implemented, I have used it successfully on several projects. On each one, I have made minor modifications to fit the shell into a particular environment. Here are some variations that could be helpful:

- A global out of memory handler can be installed in *malloc( )* and *realloc( )*. The handler can either crash the program or attempt to recover. I used this on one project and found it of limited utility. Error recovery is best handled by the memory shell clients, not a global error handler. When retrofitted to existing code, however, you can at least find out whether a problem exists.
- Overwrites of memory can be detected in a limited way by adding a few extra bytes to the start and end of each client data region. If these bytes are filled with magic values, the values can be checked when the block is freed. If a client has written outside data region, the values should be trashed. Note: Don't use either 0x00 or 0xff for your magic values as these are the most common values for "off by one" overwrite errors.
- A common memory allocation problem is the use of a pointer to memory that has been freed. The pointer looks completely valid for most purposes, but the memory pointed at no longer belongs to the program. The memory remains unchanged in most implementations, however, so the values stored by the client in the memory block will probably be valid until the block is reused by another call to *malloc( )*. This is referred to as using a stale pointer and can cause very erratic program behavior. If you suspect that stale pointers are being used, the memory shell can zero the data blocks in *mem_free( )* immediately before the *free( )*. This way, if a stale pointer is used to access data in a freed block, the data returned is zero. A variation on this is to complement all bytes in the block. By complementing the bytes, you can guarantee that the data in the block is scrambled when accessed through a stale pointer. This should cause your program to blow up quickly.
- Memory tags are not guaranteed to be unique. There is a finite probability that an arbitrary memory area can contain the expected memory tag value. If you are worried about using memory tags to check for a valid memory block, you can search the memory block list at the places where I check memory tags for validity. If you can't find the block pointer in the list, the block is invalid. Such a list search can take significant execution time, so use this approach only if all else fails.

Roll Your Own

The variations on a memory shell are endless. Every time I build a new application I find one or more improvements. The important thing is to encapsulate the memory routines. Once they are encapsulated, you will have control over the memory allocation and can add the features that your application needs. You will come up with uses that I cannot even imagine. Good luck.

DDJ

(Listings begin on page 110.)

Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

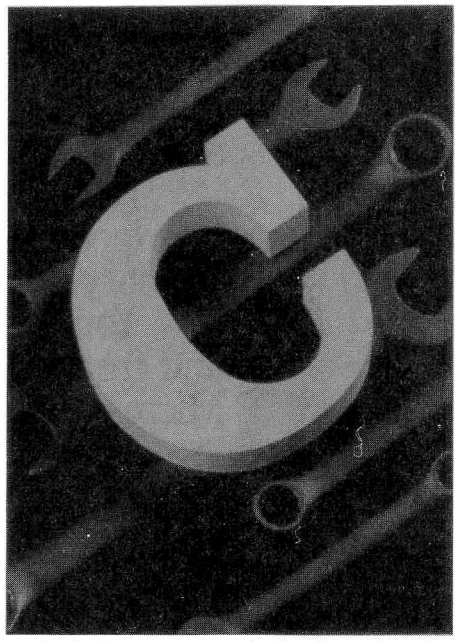
AWK as a C Code Generator

The unique features of this special-purpose language can greatly improve development time

Wahhab Baldwin

The AWK language is one of the gifts that the Unix environment has given to us. Named for the initials of its developers (Alfred Aho, Peter Weinberger, and Brian Kernighan), AWK was originally designed to be a simple utility for "quick and dirty" programming tasks. Since then, it gradually evolved into a powerful tool capable of performing many time-saving tasks for programmers.

This article will introduce the AWK language and present an AWK application that functions as a C code generator. While the specific problem solved here may not be immediately useful to you, the principles applied to solve the problem will give you ideas about how you, too, can use AWK to handle your labor-intensive tasks.



The AWK Language

The AWK language is an example of a special-purpose language. Special-purpose languages are not designed to solve every programming problem. Instead they can greatly ease the development effort required in the arena for which they are intended to be utilized. AWK was designed to be a tool for writing programs that first read one or more sequential files from standard input and then write a file to standard output. While AWK can be used for developing programs that handle other tasks, it excels at tasks of this sort.

AWK saves you time because it makes some assumptions and performs many actions without the need for you to ask that it do so. An AWK program consists of a sequence of pattern-action statements (as described shortly) and function definitions. Essentially, AWK executes the following cycle:

1. Read a line from standard input.
2. Parse the line into fields. The special variable *$0* is assigned to the whole line (without the carriage return). The special variable *$1* holds the first field, and *$n* holds the nth field.
3. Set other variables as follows: Set *NF* to number of fields, *NR* (number of record) to the line number, *FNR* to the line number within the current input file, and *FILENAME* to the name of the current input file. Thus, *$NF* holds the last field of the line.
4. Process the record according to each of the pattern-action statements in order.

A pattern-action statement takes the form: *pattern {action}*

The statement can be read in this way: If the input line matches the pattern (or condition), then perform the action. Either the pattern or the action can be omitted. If the pattern is omitted, then by default the action is to select every input line. If the action is omitted, then by default the action is to print the input line.

The syntax of AWK is very similar to the C syntax, except that in AWK a carriage return can replace C's ubiquitous semicolon. Also variables can be used in AWK without being declared, and they are initialized either to zero or a null string, depending on how

Wahhab has more than 20 years experience in software development and is currently owner of Baldwin Software Services, a small company specializing in helping large companies apply new technology and improve their software development process. He can be reached at 1011 Union St., Manchester, NH 03104.

(continued from page 36)
they are used.

A crucial feature of AWK that is not present in C is the ability to perform regular expression matching. Regular expression matching is used by the Unix *grep* utility, and has been adopted by several text editors, including Brief. AWK compares a pattern string (usually enclosed between slashes) against another string (which, by default, is the input line), and then returns true or false depending on whether the pattern is matched. (For more details, see the accompanying text box entitled "Regular Expressions.")

At this point, without explaining the command syntax in detail, we can explore some small but still useful AWK programs. For example, */Fred/* prints all lines in the specified input that contain the string "Fred". The following one-line program prints its input file with line numbers: *{print FNR, $0}*

If alignment of the input is necessary, then the *printf* statement familiar to all C programmers can be used: *{printf("%4d %s\n", FNR, $0)}*

The comma between the two parameters in the print statement causes an output field separator (a space by default) to be placed between the fields. If the space is not desired, two strings can be concatenated in AWK by simply naming them one after another (that is, *print FNR $0*). Note that *printf* requires

AWK saves you time because it makes some assumptions and performs many actions without the need for you to ask that it do so

Regular Expressions

Regular expressions use a pattern string of characters, enclosed in slashes, to describe a pattern that can be compared to a target string. The simplest example of a pattern string is just a string of characters. For example, */red/* matches "a red ball" and "Alfredo", but not "tuna fish". Different series of metacharacters have special meaning in the pattern string. The period refers to any single character, so */r.d/* matches "red" and "rod", but not "road" or "rd". To match a period or any other metacharacter literally, precede it with a backslash. Special characters can be matched by using C's escape characters: *\t* represents a tab and *\b* represents a backspace, while *\101* represents an octal value (65, or the character 'a').

A circumflex matches the beginning of a string while the dollar sign matches the end, so */^red$/* matches a line consisting solely of the word "red". An asterisk matches zero or more of the preceding pattern. For instance, */l*ama/* matches "ama", "lama", "llama", "lllama", and so on. A plus matches one or more, while a question mark matches zero or one of the preceding pattern. Also, characters may be grouped with parentheses. Thus, */M(iss)+ippi/* matches "Missippi", as well as"Mississippi". Brackets can be used to indicate a character that is one of the enclosed characters, so that */[a-zA-Z]* matches any letter, while */[13579]* matches an odd digit. Thus, */[A-Z][a-z]*/* matches any word with an initial capital. A circumflex that is the first character after the open brace matches any character except for those in the braces, so */^[^aeiou]+$/* matches strings with no lower-case vowels.

Regular expressions are a valuable and powerful tool. Unfortunately, different implementations of regular expressions occur under the MS-DOS programs that use them. The Microsoft Editor does not support parentheses or the plus symbol; the Brief editor uses braces in lieu of parentheses, the **@** to match zero or more occurrences, and **?** as a single character, and has other differences as well. Different MS-DOS implementations of *grep*, a command-line text search facility based on the Unix utility, use slightly differing characters. Nonetheless, regular expressions are a powerful tool for matching text — once you become familiar with them, you will wonder how you ever lived without them.

— W.B.

(continued from page 38)
a *newline*, while *print*, like its Basic counterpart, prints a new line automatically.

Two special patterns, *BEGIN* and *END*, refer to one cycle before the first input record has been read and after the last input record has been read. For instance, Example 1 shows how to total a series of numbers.

```
        {total += $1}
END     {print total}   # Total for all input
```

Example 1: *A simple AWK program to total a series of numbers.*

```
{gsub(/aluminium/, "aluminum")}
{gsub(/colour/, "color")}
```

Example 2: *A built-in AWK function for global substitution.*

```
        # Program to count word usage in input.
        # eliminate all non-letters except space
        {    gsub(/[^A-Za-z ]/, "")         }

        # add words to associative array
        {    for (i = 1; i <= NF; i++)
                ++word[$i] }

        # print each array entry
END     {    for (j in word)
                print j, word[j] }
```

Example 3: *Using strings rather than integers as array subscripts*

The += operator, when used as shown in Example 1, is equivalent to *total = total + $1*. Also note that the pound sign (#) indicates a comment.

These programs are typically run by first saving them as a file (say, TOTAL.AWK) and then invoking the file by the command line: *AWK -f TOTAL.AWK DATA1.DAT DATA2.DAT*

DATA1.DAT and *DATA2.DAT* serve as the input files (*DATA*.DAT* could also be used). The output goes to *STDOUT* and is typically redirected into a file by using >. Short programs can be passed directly as a parameter to AWK: *AWK '{print FNR, $0}' prog.c > progc.lst*

This last method is more useful under Unix than under MS-DOS, because MS-DOS restricts both the length of the command line and the use of < or > in a parameter.

Note that AWK is traditionally implemented as an interpreter, although an AWK compiler for DOS now exists (see the text box entitled "AWK for DOS"). Most AWK programs are small and I/O bound, so the quick debug cycle of an interpreter generally outweighs the speed benefits of a compiler.

AWK contains many useful built-in

functions for handling strings. For instance, the program in Example 2 performs global substitutions on the input file, changing British spelling to American. Other string-handling routines include *sub*, which performs a single substitution; *length* and *substr*, which return the length and substrings, respectively; and *index*, which finds the index of the first occurrence of one string as a substring of another.

AWK is unusual in that all variables are both treated either as strings or as numbers, depending on how they are used. Arrays in AWK are *associative arrays* whose subscripts are strings, rather than numbers. This powerful facility, also found in other languages such as SNOBOL and REXX, allows the construction in memory of a structure that is similar to a keyed file. Thus, the program in Example 3 produces a list of each text word used in the input file along with the number of times that the

```
struct rec1 {
    long  r1_id_no;
    char  r1_name[51];
    int   rc_value;
};
struct rec2 {
    long  r2_id_no;
    int   r2_seq;
    char  r2_code;
    struct {
        char  r2_state_cd[3];
        long  r2_state_eff_dt;
    } r2_st_range[51];
    struct {
        int   r2_value;
        char  r2_use_cd[3];
    } r2_not_array;
    struct {
        char  r2_work_cd[3];
    long  r2_work_dt;
    } r2_work_list[6];
    long  r2_the_end;
};
```

Figure 1: *Sample input for Listing One*

```
/* rec1 */
        stlong(p.rec1->r1_id_no , inf_rec + 0);
        stchar(p.rec1->r1_name , inf_rec + 4);
        stint(p.rec1->rc_value , inf_rec + 54);
/* rec2 */
        stlong(p.rec2->r2_id_no , inf_rec + 0);
        stint(p.rec2->r2_seq , inf_rec + 4);
        stchar(p.rec2->r2_code , inf_rec + 6);
        for (i = 0; i < 51; i++) {
            (p.rec2->r2_st_range[i].r2_state_cd , inf_rec + i * 6 + 7);
            (p.rec2->r2_st_range[i].r2_state_eff_dt , inf_rec + i * 6 + 9);
        }
        (p.rec2->r2_not_array.r2_value , inf_rec + 313);
        (p.rec2->r2_not_array.r2_use_cd , inf_rec + 315);
        for (i = 0; i < 6; i++) {
            (p.rec2->r2_work_list[i].r2_work_cd , inf_rec + i * 6 + 317);
            (p.rec2->r2_work_list[i].r2_work_dt , inf_rec + i * 6 + 319);
        }
        stlong(p.rec2->r2_the_end , inf_rec + 353);
```

Figure 2: *C code generated by the AWK program in Listing One*

text word occurs.

I have found AWK to be very useful during the process of converting a large system over from one database manager, such as Informix's C-ISAM, to another. C-ISAM stores data internally in an operating system-independent format. This means that each field must be converted to and from this format when records are read or written. I needed to convert many different record types, each with many fields, and the offsets for each field had to be calculated exactly. I used the AWK program in Listing One (see page 116) to generate the C code required.

In this case, my input files were C header files that contained record structures. In turn, some of these record structures contained other structures or arrays of structures. My output was C source code to move the data from these records as pointed to by a pointer called *p*. A bit of sample input and output are shown in Figures 1 and 2, respectively.

All of my header files had the same basic format, so I didn't try to make the AWK program more general. For example, the opening *struct* statement and the closing brace for each record must start in column 1, while all other lines must be indented. (After all, this was a one-shot program, designed to be used once and then thrown away.)

The first line in Listing One changes AWK's default input file separator from its usual value of white space to be any mix of spaces, tabs, or open brackets. A quirk of AWK is that when you modify the input *field separator*, the leading spaces on a line are considered to be the first field. If you use the default, the leading spaces are ignored. The field names (which are indented) show up as the second field, or as *$2*, rather than *$1*.

The next line of the program deals with input lines such as: *struct sample_rec {*, and resets *j*, which is the offset within the structure. At the same time, the record name is saved for use in the *emit* function described in the next paragraph.

The bulk of the program deals with the individual fields. If the field occurs within a structure inside of the main record structure, the information in the field must be saved until the structure name and the number of occurrences are encountered. The function *setup* does this, using AWK arrays to hold the information. In the normal case, the

```
        if (type == ''string'')
            last_part = ", " l ");"
        else
            last_part = ");"
        print "\t" type "(p." rec "->" varname
            ", inf_rec + " offset last_part
```

Example 4: *Alternative to using the* **?** *operator*

AWK for DOS

Two commercial versions of AWK are available under MS-DOS and OS/2. Mortice Kern Systems, Waterloo, Ont., Canada, includes both small and large model versions of AWK in its MKS Toolkit (with and without 8087 support), and also sells AWK separately, offering both DOS and OS/2 versions. This package includes a tutorial written by MKS, as well as a copy of the book, *The AWK Programming Language* by Aho, Kernighan, and Weinberger (Addison-Wesley, 1988). The OS/2 version opens the possibility of holding enormous arrays in memory.

Sage Software (Beaverton, Oreg.), which recently acquired Polytron, provides the same book in its versions of the PolyAwk program for MS-DOS and OS/2. PolyAwk includes several useful extensions, such as the ability to hold associative arrays in sorted order, the provision of true multidimensional arrays, and the availability of new functions including *getkey*, *toupper*, and *tolower*.

Sage also offers a developer's toolkit that includes both an interpreter and a compiler. By default, the compiler, AWKC, produces a program with an extension of .AE. This program must run by a run-time program, called AWKI. However, by using a *-xe* flag on the AWKC command produces a stand-alone executable program. The ability to write stand-alone programs using AWK is a great benefit if you wish to distribute your software to others — they do not need a copy of the interpreter, and they cannot read or modify your source code.

— W.B.

(continued from page 42)
emit function is called. This function drops any trailing semicolon from the variable name (from a line such as *int field;*), and then prints the line of C code required. The *print* statement in *emit* uses the C operator **?** to print the second parameter required by strings. A simpler but longer approach would require the use of an extra variable as shown in Example 4.

AWK does not have function pointers. I wrote the *f* function to invoke the appropriate function and to pass on the remaining parameters.

The last remaining task occurs when the closing brace to a structure within a structure is found. If *NF* > *3*, then this is an array of structures, and a C language *for* loop is printed. The field separator includes the open bracket but not the close bracket, so in a line such as, : } inner_struct[5]; *$4* is set to *5];*. The trick *$4 + 0* forces this to a numeric value.

AWK does not have function pointers

The AWK *for* loop uses the temporary variable *k* and then calls the function *emit2*, which is a close copy of *emit*, except that it calculates an offset from *inf_rec*. When *emit2* has spit out one line for each variable in the inner structure, it prints the closing brace to the C *for* loop, and calculates an updated offset into the C-ISAM record.

This AWK program handles the process of moving data from the C structure to the C-ISAM record. I wrote an almost identical program to handle the reverse process of moving the data from the C-ISAM record to the C structure. The only problem that had to be addressed differently in the second program was how to handle slack bytes. Different C compilers place slack bytes in various places to encourage numbers and structures to be word-aligned. Based on the conditions that your particular compiler and settings require, the addition of a line such as this will force the offset to a word boundary: *offset += offset % 2*

While this program has obvious weaknesses (which I would remedy if it were to be run by others), it still serves as a good example of how to use a utility language such as AWK to perform time-consuming chores that could take days if done manually, and that would even take much longer to program in a conventional language such as C. In fact, having a powerful tool such as AWK handy will lead you to do many tasks you might not undertake otherwise! Now that an AWK compiler is available (see "AWK for DOS"), you may even find yourself using AWK for distribution programs.

Source Code Availability

As a service to our readers, all source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Drive, Redwood City, CA 94063, or call 800-356-2002 (inside Calif.) or 800-533-4372 (outside Calif.). Specify issue number and disk format. Code is also available through the *DDJ* listing service (415-364-8315) and through the DDJ Forum on CompuServe (type GO DDJ).

DDJ

(Listing begins on page 116.)

Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

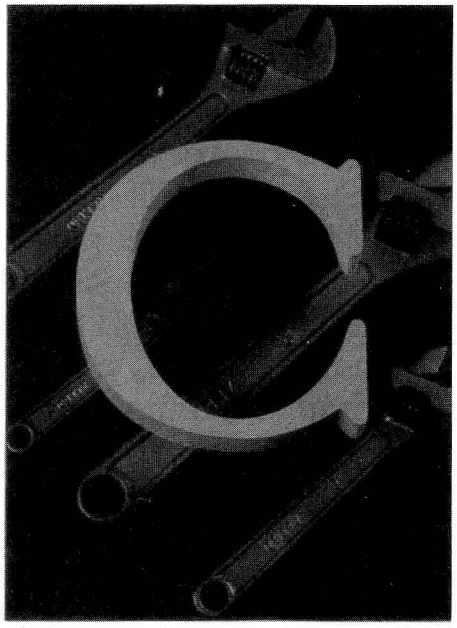
Implementing Bicubic Splines

Drawing objects that contain curves

Raymond G. Lauzzana and Denise E.M. Penrose

Lisp is an excellent language for developing high-level AI programs. In particular Allegro Common Lisp (Allegro CL) has made the Macintosh an excellent platform for developing AI software. All implementations of Common Lisp including Allegro CL, are limited by some serious drawbacks. Communications to graphics and audio devices are not specified and Common Lisp does not even define access to physical devices. (A trap mechanism in Allegro CL does permit communication to the Macintosh Toolbox, but this mechanism is very limited.) Additionally, complex mathematical computations, such as geometric transforms or statistical analysis, are extremely slow in any implementation of Lisp.

To overcome these problems, you can write C functions that are called from Lisp. Allegro CL provides a Foreign Function Interface to facilitate interlanguage communication. The use of this interface allows C to access physical devices and perform complex computations on behalf of higher-level Lisp operations. This allows the Lisp program to be concerned with high-level problems such as rule-base management, while the C routines handle all of the "dirty work" such as drawing pictures on the screen.



In this article, we present a spline function that uses the Macintosh Toolbox to draw a smooth curve. This spline function is one of 18 graphic primitives in Artifex, a design-rule system based on shape grammars. (Source code for the system is available from the authors of this article.) In order to calculate the points along the curve of a bicubic equation must be solved. Calculations of this sort are miserably slow in Lisp. However, the solution presented here is quick enough to reasonably support user interaction, such as "rubber-splining," in which the user dynamically instances splines.

The Bicubic Spline

"Splines" are graphic elements that are used for drawing objects that contain curves. A spline is composed of one or more "spline segments." Figure 1 illustrates a spline segment defined by the four points *P0* though *P3*. Points *P1* and *P2* are the end points of the spline segment and the curve is drawn between these two points. Points *P0* and *P3* are called the "control points." During the process of calculating the spline, the two points are interpreted as vectors.

The magnitude of the vector modulates the tension of the curve. In other words, the greater the distance between *P0* and *P1*, the more the curve will bend. As Figure 2 shows, when *P0* and *P1* are coincident, the beginning of the curve is parallel to the line *P1 – P2*. As *P0* approaches infinity, moving further away from *P1*, the curve tends toward

Raymond is a research associate at the Center for Knowledge Technology, Utrecht, The Netherlands. He is the developer of various software for research in design rule systems, color semantics, and image analysis. He also teaches Lisp and AI at the Hogeschool voor de Kunsten Utrecht. Denise is a freelance writer and editor covering AI and electronic art in Europe. Previously, she was principal editor at Lotus Development Corp. and senior editor at Osborne/McGraw-Hill. They can be reached at Oudegracht 317, 3511 PB Utrecht, The Netherlands, telephone: +31 (30) 340 866.

(continued from page 48)
the line *P0 – P1*. Similarly, the spline may be bent to one side or the other as illustrated in Figure 3, by modulating the vector angle. In this manner, a user can interactively describe a curve by moving its control points in order to perform the rubber-splining mentioned earlier.

The spline presented here uses a kind of interpolation known as "bicubic" to create a particular class of curves. The coordinates for the points, (*i*, *j*), along this curve are found by solving the cubic equations shown in Example 1.

$$j = j_1t + j_2(1 - t) + W(j_2 - j_3)(1 - t)^2t + W(j_1 - j_0)(1 - t)t^2$$
$$i = i_1t + i_2(1 - t) + W(i_2 - i_3)(1 - t)^2t + W(i_1 - i_0)(1 - t)t^2$$

Example 1: *Cubic equations*

To draw a curve, you must solve this pair of equations for every point at some regular sampling interval. The number of samples determines the coarseness or the smoothness of the curve. Figure 4 illustrates a spline segment that results from sampling the equation four times. If you sample the equation more frequently, your curve will be smoother. The production of a reasonably smooth curve requires a fairly short interval with a width of 2 – 3 pixels and a significant amount of computation.

The free cubic variable *t* has a range from 0 to 1 in real numbers. To find a point in the middle of the curve, you solve the equations for *t=0.5*. In this way, the degree of smoothness of a curve can be varied, depending on the frequency at which *t* is sampled.

The other free variable, *W*, is a weighting factor that controls the tension of the curve. A value of 0.9 produces reasonable curves on the Macintosh. If *W* is 0, then a straight line is drawn between *P1* and *P2*. As *W* increases, the end of the curve tends toward tangency with the lines *P0 – P1* and *P2 – P3*. Figure 5 illustrates the effect of varying this weighting factor.

It is important to remember that an infinite number of curves can be drawn between any two points. Bicubic interpolation produces curves with continuities at the end points. These continuities facilitate the process of combining spline segments into an infinite class of curves.

Figure 6 illustrates two bicubic spline segments. The first spline segment is defined on the line *P0-P3* and the second defined on the line *P1-P4*. Together the two segments form a spline

Help Functions

One C routine (shown in Listing One) and four Lisp Help functions (presented in Listing Two) are required to calculate and display the spline.

The C routine *distance* calculates the mean-squared distance between two coordinate pairs. This calculation is used in *_Spline* to determine the distance between the end points.

A higher-level Lisp representation for a point @@ replaces the intrinsic #@ provided with Allegro CL. This representation establishes device-independence for high-level functions, such as *SPLINE*. The function @@ describes a screen location in physical coordinates. *(@@ 0 0)* is the upper-left corner. If a single integer is received, the integer is interpreted as a packed point, and it is unpacked into a coordinate pair.

The Lisp predicate *POINTP* tests for the pointness of its argument. If the argument is a point represented in coordinates *(@@ 00)* the argument returns *T*; otherwise the argument returns NIL. The two functions *I-COORD* and *J-COORD* extract the vertical and horizontal coordinates from a point.

— R.L., D.P.

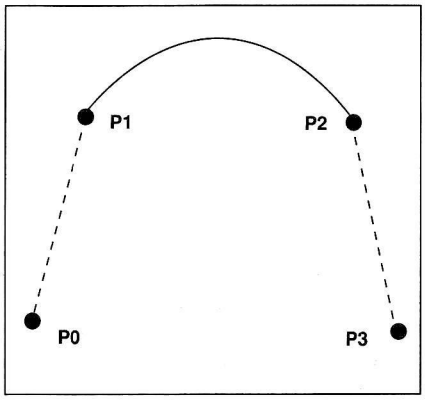


Figure 1: *A spline segment*

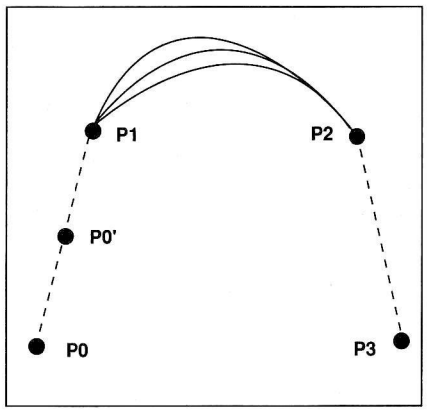


Figure 2: *Controlling the vector distance*

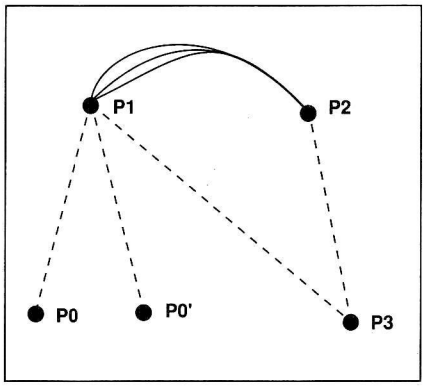


Figure 3: *Controlling the vector angle*

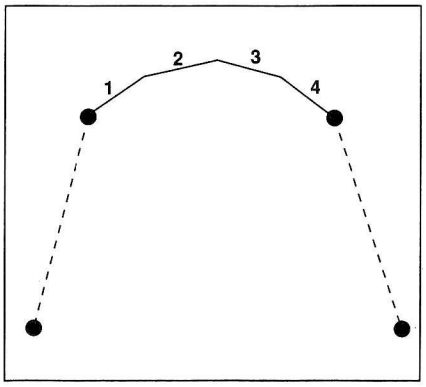


Figure 4: *A spline segment drawn into four samples*

that is continuous in the second derivative at *P2*. In other words, the two spline segments change at the same rate and have the same slope at *P2*, so they connect smoothly. This special property of bicubic splines makes them ideal for use with graphics software because a smooth curve can be drawn through any arbitrary set of points. Another advantage of bicubic splines is that the points that define the curve of the splines lie on the curve. Other curve-defining functions, such as the B-spline, are defined by points that are not located on the curve.

You could write a Lisp routine to calculate the points along the curve, but such a routine would be very slow. In any case, you would still need to access a physical line-drawing function in order to connect the points into a curve. The faster and better way to handle this calculation is to call the C routine, *_Spline*, which is found in Listing One (page 118). This routine is accessed through three Lisp functions: *Mac_Spline*, *Do_Spline* and, at the highest level, *Spline*, found in Listing Two (page 119).

The C Routine

The routine *_Spline* calculates and draws a curved line along the path of a bicubic spline. This routine must solve the equations mentioned earlier for each point along the curve. The parameters to *_Spline* are the coordinates *(i,j)* of the four points *P0*, *P1*, *P2*, and *P3*. The routine has two parts: an initialization phase and a loop. Speed is important, so the routine contains the minimum number of repetitive calculations.

The initialization phase sets up a two-pixel-wide sampling rate, and a weighting factor of 0.9. If you wish to create faster but coarser splines, you would increase the value of *inc* to either 5 or 7. After these values have been established, the Macintosh Toolbox function *MoveTo* sets the current position to P1. The routine then calculates the distance between *P1* and *P2*. If that distance is less than the sampling interval, *_Spline* uses *LineTo* to draw a straight line from *P1* to *P2*.

To increase the speed of the routine, the bicubic equation is partially solved outside of the loop. A straightforward calculation of the coordinates would result in this loop with 18 multiplies and 19 additions, (Example 2).

A more efficient partitioning of the equation produces this loop with 13 multiplies and 8 additions, as shown in Example 3.

The reduction in additions results from precalculating the factors, *i3*, *j3*, *i5*, and *j5*. The reduction in multiplication results from calculating the cubic factors *t2*, *t3*, and *t5* only once for both coordinates. This may seem like a relatively minor improvement, but the calculation must be performed 100 or more times while the user is trying to instance a curve. In this sort of situation, every nanosecond counts.

The cubic variable *t* has a range from 0 to 1. The increment on this unit interval, *dt*, describes the rate of sampling of points on the spline. *dt* is the unit increment (Δt) on the variable *t*. For example, if you want to sample a point every 2.5 pixels and the distance between *P1* and *P2* is 100 pixels, the unit increment *dt* would have the value of

```
while (t < 1)
{ j = j1*t  + j2*(1 - t)  + W*(j2-j3)*t*(1 - t)*(1 - t)  + W*(j1-j0)*t*t*(1 - t);
   i = i1*t  + i2*(1 - t)  + W*(i2-i3)*t*(1 - t)*(1 - t)  + W*(i1-i0)*t*t*(1 - t);
  t = t + dt;
  LineTo(i,j);
}
```

Example 2: *Calculating coordinates*

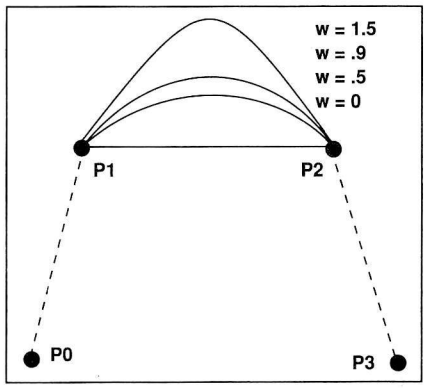


Figure 5: *Varying the weighting factor*

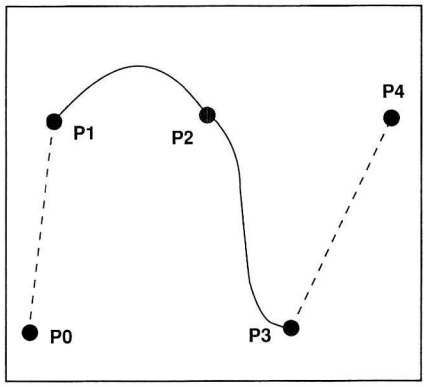


Figure 6: *Two spline segments forming a continuous curve*

0.025. The reciprocal of *dt*, *count*, which is the number of points to be calculated. In this case, 40 points would be sampled along the spline.

In order to improve accuracy, the loop in the actual code is incremented with respect to *count* rather than *t*. This step incurs the additional cost of one multiply, but the resulting elimination of round-off errors improves the appearance of the curve.

Substantially worse algorithms that do not factor out the cubic variable *t* can be used to solve this equation. Better algorithms are also probably available. When working on problems of this sort, reach into your bag of algebraic tricks before you start coding.

Loading the C Routine

Before you can use the C routine from Lisp, you must first compile the routine and then load the Allegro CL Foreign Function Interface into your Lisp environment. Remember that you can only load relocatable object files, not sources or programs. In other words, you may only load compiled subroutines. The Foreign Function Interface will load any MPW C relocatable object files. It will not work with LightSpeed C or other third-party compilers.

Once you have compiled your C routine, start up Allegro CL. The simplest way to load the Foreign Function Interface uses the expression *require*. In order to use this expression, you must have already moved *FF.fasl* from the Foreign Function folder to the Library folder, as shown in Example 4(a).

Another way to load the Foreign Function Interface uses *load*. In this case, you'll need to specify the exact *<pathname>* to the folder where you have stored *FF.fasl*. See Example 4(b). You will also need to specify the folder that stores the C library functions that are provided with MPW C. To do so, create a logical pathname *CLIB* using *def-logical-pathname*, as demonstrated in Example 4(c).

You are now ready to load the *_Spline* function into your Lisp environment. The Foreign Function Utility *ff-load* loads binary files and their associated libraries. See Example 4(d).

```
while (t1 <  1)
{ t1 = t1 + dt;
   t2 = 1.0 - t1;
   t5 = W*t1*t1*t2;
   t3 = W*t1*t2*t2;
    j = j1*t1 + j2*t2 + t3*j3 + j5*t5;
    i = i1*t1 + i2*t2 + t3*i3 + i5*t5;
    LineTo(j, i);
}
```

Example 3: *Partitioning the equation shown in Example 2*

(continued from page 53)

In this case, the object module for the function *_Spline* is stored in the file called "spline.c.o." *ff-load* searches this file for the symbolic name *_Spline*. If it finds the name, it loads the associated binary code. *ff-load* then searches the libraries for any unsatisfied symbolic references, which may have occurred within the code, and loads them as well. For example, if you use the C *sqrt* function, you must tell Allegro CL which library to look in for the function. In other words, *ff-load* links and loads the routines identified in the *:entry-names* list, using the libraries in the *:libraries* list. If you wish to load more C functions, add them to the *:entry-names* list. *ff-load* is a time-consuming function, so you should call it as infrequently as possible.

Binding the C to Lisp

Once you've loaded *_Spline* into your environment, the function's code resides in the dynamic memory of your machine. To use the function, you must establish a symbolic reference to its code by binding it. In this case, the symbolic reference *Mac_Spline* is bound to the C symbol *_Spline*.

In Allegro Lisp you can bind a C function in two ways. The first method uses *deffcfun*. *_Spline* has eight parameters: *i0*, *j0*, and so forth. Each of the parameters is an integer and must be declared as such. The flag *:novalue* indicates that the routine produces only a side effect; the routine does not return a value. Types for both input and output variables are declared in this manner. Example 5(a) demonstrates this method.

Though *deffcfun* is a more simple function, the binding is lost if you build a stand-alone application. Most of the C variable types, such as *long*, *char*, *integer*, and *word* are available using *deffcfun*, but if you want to pass a list from Lisp or a structure from C, you are on your own. Good luck!

The second and preferred method of binding a C function uses *ff-look-up-entry*, *ff-call*, and *multiple-value-bind* to bind the physical entry point to a Lisp symbol. This method can be used for any MPW C relocatable object code, not just for C functions. In other words, you could even bind assembly language

```
(a)

(require 'FF)

(b)

(load "<pathname>:FF.fasl")

(c)

(def-logical-pathname "CLIB;"  "<pathname>:MPW:CLibrary")

(d)

(ff-load "<pathname>:spline.c.o"
         :entry-names
         (list "_Spline" )
         :libraries
         (list "CLIB;StdCLib.o" "CLIB;CRuntime.o" "CLIB;CInterface.o"
                "CLIB;math.o" "CLIB;CSANElib.o"))
```

Example 4: *Loading the C routine*

or Pascal code by using this method. This method is a hardware-dependent solution, because you must represent the binding in terms of the physical size and order of the machine's stack. In the case of the Mac II, parameters are stored in reverse order on the stack. The foreign function *call* must push the parameters onto the stack in reverse order.

```
(a)

(deffcfun (Mac_Spline "_Spline")
 (integer integer integer integer integer integer integer integer) :novalue)

(b)

(defun Mac_Spline (a b c d e f g h)
 (multiple-value-bind (entry a5) (ff-lookup-entry "_Spline")
      (ff-call entry
       :a5 a5 :long h :long g :long f :long e :long d :long c :long b :long a :novalue)))

(c)

(Mac_Spline  3  4 120  140  300  260 490 600)
```

Example 5: *Binding C to Lisp*

The function *ff-lookup-entry* finds the physical address of the entry point for the *_Spline* routine and the location of the register to be used when the routine runs. Because this function returns two values, you must use the some-

These functions allow you to link C with Lisp for accessing physical devices and improving the computational performance of your Lisp environment

what esoteric Common Lisp function *multiple-value-bind* to assign these values to the local variables *entry* and *a5*. The function *ff-call* then executes the code stored at the physical address specified by *entry*, using register *a5*. Notice that the parameters in *ff-call* are reversed from the declaration of *Mac_Spline*, as shown in Example 5(b).

At this point, Lisp can call your C routine, *_Spline*, through *Mac_Spline*. If you typed the following example into the Listener window, a curve would be drawn between the points *(@@ 120 140)* and *(@@ 300 260)*. The control points for the curve are *(@@ 3 4)* and *(@@ 490 350)*. The result is shown in Example 5(c).

Mac_Spline might serve as an adequate level of representation for some problems, but in the case of the spline, a higher level of representation is needed.

The High-Level Lisp Functions

The *Mac_Spline* function draws a spline segment in the current window. Because the user probably wants to combine several spline segments into a continuous curve, a high-level function *Do-Spline* must be used instead. *Do-Spline* is a tail-recursive function, and its argument is a list that contains an arbitrary number of points. In other words, *Do-Spline* draws a spline based on the first four points, then the next four points, and so on, until only three points remain in the list.

(continued from page 56)

In this example, *Do_Spline* draws two spline segments connected at the point *(@@ 300 260)*. See Example 6. The first spline segment is identical to the spline segment drawn in the previous example. The second spline segment is a curve between the points *(@@ 300 260)* and *(@@ 490 350)*. The two curves are continuous at *(@@ 300 260)*. A few problems still remain. First,

"Splines" are graphic elements that are used to draw objects containing curves

Do_Spline draws the curve in your Listener window. You probably don't want to draw the curve over the text in the Listener window. Instead, you'd like to select the window in which the curve will be drawn. It's also preferable to invoke the function by using the points themselves, rather than by using a list that contains the points. The highest-level Lisp function *SPLINE* provides these capabilities:

(SPLINE window p0. . . . pN)

SPLINE uses the Allegro CL object function *with-port* to select the graphics port in which the spline will be drawn. It then calls *Do-Spline* to actually draw the spline. As a result, the *SPLINE* function can be called with any number of points and targeted to any window. This function could be instanced in a display list, referenced from a menu, and otherwise integrated into a graphics package.

General Usage

This spline function can be modified and used as a template for facilitating general connectivity between Lisp and C, or for writing similar C functions for use by Lisp. The Allegro CL Foreign Function Interface includes six functions to support this effort: *ff-load, deffcfun, deffpfun, ff-call, ff-lookup-*

```
(Do-Spline (list (@@  3  4) (@@ 120  140)
(@@ 300  260) (@@ 490 350) (@@ 25 30)))
```

Example 6: *Drawing spline segments with Do_Spline*

entry, and *dispose-ffenv*.

ff-load loads any MPW C object file and can be used to load assembly or Pascal code, as well as C code. It returns a foreign function environment that consists of code segments, a jump table, a static data area, and a set of active entry point names. It also removes dead code so that the environment includes only the code and the data that can be reached from the active entry points.

deffcfun and *deffpfun* define functions that coerce and type-check arguments before calling the foreign function. *deffcfun* is used for C and *deffpfun* for Pascal.

ff-call transfers control to the loaded object code and passes arguments according to the type-keyword/argument pairs. Possible type-keywords are *:word*, *:long*, *:ptr*, the data registers *:d0* to *:d7*, and the address registers *:a0* to *:a5*. It is a low-level function that is faster than *deffcfun* and *deffpfun*, because it does not type-check or coerce arguments. Its return values are indicated by similar keywords *:word*, *:long*, *:ptr*, *:d0* to *:d7*, *:a0* to *:a4*, and the empty value *:novalue*.

ff-lookup-entry returns two values that describe the entry point. The first value is a pointer to the entry point. The second value is the *:a5* address register pointer for the environment where the entry point was found. *ff-look-up* returns NIL if the specified entry-point name doesn't exist. If the entry-point name exists in more than one environment, the function returns an undefined environment.

dispose-ffenv unloads the foreign function and frees up the memory space that was allocated for your functions.

These six functions are the tools that allow you to link C with Lisp. Together they provide a mechanism for accessing physical devices and for improving the computational performance of your Lisp environment.

Product Information

MPW C
Apple Programmers and Developers Association (APDA)
20525 Mariani Avenue, M/S 33G
Cupertino, CA 95014
AppleLink: APDA
408-562-3910
800-282-2732 (U.S.) or
800-637-0029 (Canada)
Suggested Retail price: $150 (Fall 1989)
Requirements: MPW Development Environment, at least 2-Mbyte RAM, 128K ROM, a hard disk
Macintosh System Software 6.02 or higher

Macintosh Allegro Common Lisp (Allegro CL)
APDA (see MPW C above) Suggested Retail Price: $495 (Fall 1989)
Requirements: Macintosh Plus SE, SE/30, II or IIx
A second 800K disk drive
At least 1-Mbyte RAM

Artifex
Oudegracht 317
3511 PB Utrecht
The Netherlands
+31-(30)-340-866
Price: $300 for source code
Requirements: Macintosh Plus SE, SE/30, II or IIx, Allegro CL
A second 800K disk drive
At least 2-Mbyte RAM

References

D. Hearn and M.P. Baker. *Computer Graphics*. Englewood Cliffs, N.J.: Prentice-Hall International Editions, 1986. A good general graphics text with a clean description of a spline function.

Bartels, Beatty, and Barsky. *An Introduction to the Use of Splines in Computer Graphics*. Menlo Park, Calif.: Morgan Kaufman, 1988. A survey text that describes many types of splines.

Y. Fletcher and D.F. McAllister. "Automatic Tension Adjustment for Interpolatory Splines." *IEEE Computer Graphics and Applications*. 10, no. 1 (1990). A recent article that describes how to naturally smooth sections of the curve.

Ian E. Ashdown, "Curve Fitting with Cubic Splines," *DDJ*, September 1986.

DDJ

(Listings begin on page 118.)

Vote for your favorite feature/article.
Circle Reader Service **No. 4.**

Extending *printf()*

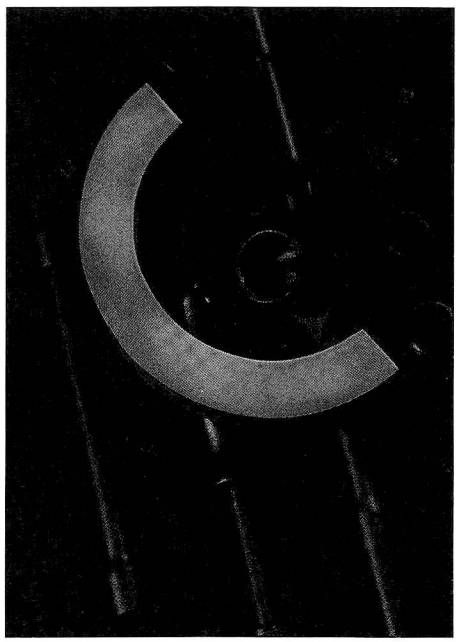
Taking advantage of variable argument lists in C

Jim Mischel

While writing my first serious financial program in C (I'd previously used the language only for utilities), I was distressed by the lack of suitable output formatting — specifically, comma-separated numbers with trailing signs. After working with Cobol financial applications several years, I expected such niceties from any language.

Faced with this shortcoming, I immediately began working on a function that provided Cobol-like output formatting using Cobol edit picture strings, and so on. The output function worked well, but was awkward to use. I wanted an easy-to-use function such as *printf( )* that would allow me to output formatted numbers as well as all the other standard *printf( )* types. I posed this problem to several C programmers and got a variety of answers. The most common answer was, "buy the runtime library code and modify *printf( ).*" I agreed that this was probably the best solution for my particular situation; nevertheless it would have been implementation-specific, expensive (Turbo C library source code costs $250), and no solution at all for compilers for which the runtime library source code is not available.

The second most common reply was, "write your own *printf( ).*" I understood that this is a common pastime among bored computer science majors and back room hackers, but I had neither the time nor the desire to reinvent the wheel in order to make a minor engineering change.



Finally, one programmer I spoke with suggested that I write a front end to *printf( )* and use this function to prescan the argument list, handling the special formats before passing the arguments to *printf( ).*

Jim is a former financial systems programmer and data processing consultant. He can be reached at 13610 N. Scottsdale Rd., Suite 10 – 251, Scottsdale, AZ 85254 (CompuServe: 73717,1355).

Variable Arguments

One of the niceties C offers is the ability to write functions that accept a variable number of arguments. The standard functions *printf( )* and *scanf( )* are the primary examples. Normally these functions have a fixed minimum number of parameters and a variable number of arguments, the number of which is usually specified either explicitly or implicitly by the fixed arguments.

There are actually two types of functions that accept a variable number of arguments. The way these work is best described by examining the prototypes for the two types of functions. The functions *printf( )* and *vprintf( )*, defined by the prototypes shown in Example 1, perform almost the same function. The only difference is how they access arguments passed to them: *printf( )* expects an unknown number of arguments to follow the format string, and *vprintf( )* is passed a pointer to a contiguous block of memory that contains the number of arguments specified in the format string.

The arguments to *printf( )* are read from the stack after being pushed in the function call. *vprintf( )* reads the arguments from the block of memory to which the *ap* variable points. On the Intel 80x86 processors, these functions work almost identically — the only difference is that *printf( )* must first determine where the block of memory starts, which just happens to be the address contained in the *SS:SP* registers. Memory maps for these function calls are shown in Figure 1.

The ANSI standard for the C language calls for a standard header file,

(continued on page 63)

(continued from page 60)
stdarg.h, that defines the *typedef va_list* and the functionality of the macros *va_start*, *va_arg*, and *va_end*. These macros are used in functions that accept a variable number of arguments. In Turbo C, the content of this header file, shown in Example 2, makes me wonder if "..." is a valid variable name.

The *va_start* macro stores initial context information in the variable designated by the pointer *ap*. *parmN* is the name of the last argument that you declare. *parmN* must not be an array type, function type, type float, or an integer type that is changed when promoted (that is, *char*). This macro must be invoked before the initial *va_arg* macro is used.

The *va_arg* macro returns the value of the argument of type *type* that is pointed to by the pointer *ap*. It then increments the pointer *ap* to the next argument.

The *va_end* macro is used to perform the cleanup operations required before the function can return. Although this macro does nothing in Turbo C, some implementations may require it, so you might want to include it in the interest of portability.

The *%m* Format Type

The *%m* format that I added to *printf( )* and associated functions outputs a comma-separated number with optional leading or trailing sign. Field output width, precision, and justification can also be specified as in the other *printf( )* formats. Default width, precision, and size, and a full explanation of the ef-

```
struct Person {
   char *Name;
   int Age;
} Jim = {"Jim Mischel", 29};

(a) printf ("%s is %d years old.\n",
Jim.Name, Jim.Age);

        Jim.Age
        Jim.Name
        &(format string)
        return address

(b) vprintf ("%s is %d years old.\n", &Jim);

        &Jim
        &(format string)
        return address
```

Figure 1: *Stack contents for* printf( ) *and* vprintf( ) *calls that output the information contained in the Person structure.*

```
int printf (const char *format, ...);
int vprintf (const char *format, va_list ap);
```

Example 1: *The functions* printf( ) *and* vprintf( ) *are defined by these prototypes.*

```
typedef void    *va_list;
#define va_start(ap, parmN)      (ap = ...)
#define va_arg(ap, type)         (*((type *)(ap))++)
#define va_end(ap)
```

Example 2: *Contents of the Turbo C header file*

fects of flag characters, are shown in Table 1.

The first major stumbling block was redirecting *printf( )* and associated function calls so that they could take advantage of my new output format. Fortunately the C preprocessor came to the rescue with macros that did the redirection for me.

These macros simply change any *printf( )* reference to *AltPrintf( )*, and *vprintf( )* to *AltVprintf( )*, and so forth. Listing One (page 120), which shows mfmt.h, the header file that contains these macros, must be included AFTER *stdio.h* in any program that uses this new format.

The *Alt . . .* functions simply set up the argument pointer, call the *Formatter( )* function to handle the *%m* formats, and then call the appropriate routine to handle the rest of the print formatting using the new format string and argument list.

Listing Two (mfmt.c, page 120) contains the code that performs the number formatting and associated functions. This code can be broken down logically into two sections: The alternate entry functions (*AltPrintf, AltFprintf, AltSprintf, AltVprintf, AltVfprintf, AltVsprintf*), and *Formatter( )* and associated functions.

The alternate entry functions provide a path between the application program and the *Formatter( )* function. The alternate entry functions intercept the arguments, set up a new argument area if necessary, and produce the final output after the *Formatter( )* function has handled any *%m* formats. These functions are never directly referenced in the source code. Instead, they are addressed through the redirection macros provided in mfmt.h.

AltVprintf, AltVfprintf, and *AltVsprintf* call the *InitBlock( )* function to allocate a temporary memory buffer for the new argument list. This new argument area is necessary because the calling program's argument block cannot be modified without side effects. The other three functions do not require this temporary block because their arguments are passed on the stack — changing them will not affect the calling program.

I wanted an easy-to-use function such as printf( ) *that would allow me to output formatted numbers as well as all the other standard* printf( ) *types*

All of the alternate functions call the *FreeBlock( )* function before exiting. This routine frees the temporary argument area and the memory allocated by the new format string (*t*).

The heart of the mfmt module is the *Formatter( )* function. After allocating the memory for the new format string, this function simply copies the old format string to the new format string, replacing all *%m* formats with the for-

Flag	How it affects output
-	Left-justifies the format
+	Generates a plus sign for signed values that are positive.
blank	Generates a space for signed values that are positive.
#	Generates a trailing sign if required by sign or one of the other flags (+ or blank).
0	Ignored
Width	Same as for "f" format
Precision	Same as for "f" format with the exception that default precision is 2, rather than 6 decimal places.
Input-size modifier	**How arg is interpreted**
none	arg is interpreted as a float
l	arg is interpreted as a double
L	arg is interpreted as a long double
Turbo C departs from the ANSI standard in how the size modifiers are interpreted. The standard specifies that "f" is a double and "Lf" is a long double. There is no "lf" format.	

Table 1: *The behavior of the new* "%m" *output format for* printf( ) *and related functions.*

matted number. Any time a % symbol is found in the format string, it calls the *ParseFormat( )* function for future processing. Upon exiting this function, *t* points to the new format string and the arguments are correctly placed in either *NewBlock* or on the stack, depending on which alternate entry function was called.

When *Formatter( )* finds a % character in the format string, it calls the *ParseFormat( )* function to perform further processing. This function picks up and saves the flags, width, precision, and size specifiers, and copies the arguments to the modified argument block.

The operation of this function and the *ParseFlags*, *ParseWidth*, *ParsePrecision*, and *ParseSize* functions are straightforward. Only a couple of items warrant mentioning.

The *ParseFlags* function uses a bit-encoding technique for reporting which flags are specified. Each of the four possible flags is assigned a power of 2. If that particular bit is set, it indicates the corresponding flag was given in the format string.

The *ParseWidth* and *ParsePrecision* functions modify two variables each: a number and a flag. The flags *WidthFlag* and *PrecisionFlag* specify which kind of width or precision was given, and the number *Width* and *Precision* specify the field width or precision, respectively.

ParseSize modifies the *Size* variable, which reports if any of the valid input size modifiers were specified.

The *ParseType* function determines which conversion type is specified and acts accordingly. Unless a *%m* conversion is specified, all flag, width, precision, and size variables are ignored, the arguments are copied directly, and no modification is done to the format string. Only when the *%m* conversion is given does any further processing occur.

Throughout the *Parse* . . . functions, I used the *AddBlockArg* macro to copy arguments from one parameter block to the other, incrementing the *NewArgPtr* variable in the process. This macro references the *va_arg* macro to retrieve the next argument from the list. The *RemoveBlockArg* macro is used to decrement the *NewArgPtr* variable so that it points to the previous argument, effectively removing that argument from the argument list.

When a *%m* conversion is identified by *ParseType( )*, the *mFormat( )* function is called to format the number into the modified format string, and to clean up stray arguments from the argument list. *mFormat( )* removes any width or precision variables from the argument

list, determines the sign of the number, and calls *dtoa( )* (Double to ASCII) to format the number into the temporary string variable *Num*. When *dtoa( )* returns, *mFormat( )* places the sign, justifies the number, and adds the formatted result to the modified format string *t*.

dtoa( ) is a fairly simple recursive number formatter. It handles precision and comma-separating, but leaves sign placement and justification to the following routines. *dtoa( )* will not correctly handle a negative number; *mFormat( )* must determine the proper sign and pass the absolute value of the number to *dtoa( )*.

There are actually two types of functions that accept a variable number of arguments

Some Notes

The mfmt code is not reentrant. I had originally planned on making it reentrant, but soon found out why *printf* and others like it are not reentrant in most implementations: Too much bookkeeping going on! As a general rule, the use of global variables is not good programming practice. The code could be made reentrant by using double indirection rather than static variables, but this would complicate the code and make it much slower than it already is.

These functions are slower than the standard *printf( )* functions. In effect, both the format string and the argument list are scanned twice, and plenty of time is spent allocating memory and shuffling things around so that the formatting will work correctly.

The functions in the *mfmt* module are a safe and portable way of modifying the behavior of the *printf( )* family of functions. However, when economic factors and the need for speed warrant it, (and portability is not a consideration) the preferred solution is to obtain the runtime library code and directly modify the *printf( )* functions.

DDJ

(Listings begin on page 120.)

Vote for your favorite feature/article.
Circle Reader Service **No. 5.**

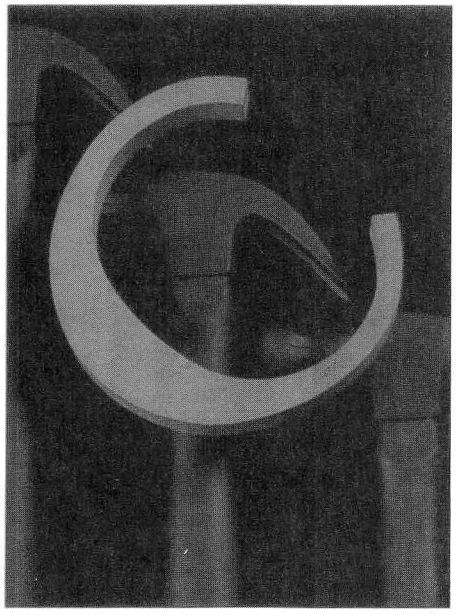
Parallel Extensions to C

Programming parallel networks

Graham K. Ellis

All extensions to C compilers for transputers are based on the Occam concurrency model. However, there are two different approaches for extending C for concurrency; one that adds keywords to the C language definition, and another that implements parallelism using library routines. Arguments can be made for either case. Bjarne Stroustrup, for example, believes that concurrency should be implemented using libraries because the library approach provides the flexibility required to implement concurrency at a lower level, closer to the machine (Stevens, 1989). On the other hand, Narain Gehani, et al. (of Concurrent C fame), hold that extending C (and C++) by adding keywords is the best solution because extensions can more adequately provide and implement the necessary concurrency functions across a wide range of platforms (Gehani and Roome).

My subjective (and completely unsupported) opinion is a politically safe "somewhere in between." I like the use of C language extensions for readability, but I think libraries allow more flexibility for different parallel architectures (important portability issues notwithstanding). But because most of the programming I do is for small real-time control and signal processing applications, what's important to me isn't necessarily important to programmers working on large programs.



On a day-to-day basis I use Logical Systems' C (LSC) compiler, an ANSI C compiler that implements the transputer parallelism using library calls. The transputer-specific library functions can be grouped into seven categories: channel communication, channel status testing, transputer concurrency (CSP model), transputer concurrency (fork/join model), transputer semaphore support, transputer timing and scheduling, and miscellaneous routines (see Table 1). Because of space constraints, I'll discuss only the channel, channel status, and CSP concurrency functions.

Library Functions

Channel communication is implemented by first allocating a channel, then reading and/or writing from process to process. *ChanAlloc( )* is used to allocate an internal channel by returning a pointer to the allocated channel. For channels assigned to actual hardware links, a pointer assignment to the hardware address is used instead of *ChanAlloc( )*. These and other transputer specifics are defined in the conc.h include file. When an allocated channel is no longer needed, memory can be freed using *ChanFree( )*. Code showing basic channel usage is listed in Figure 1.

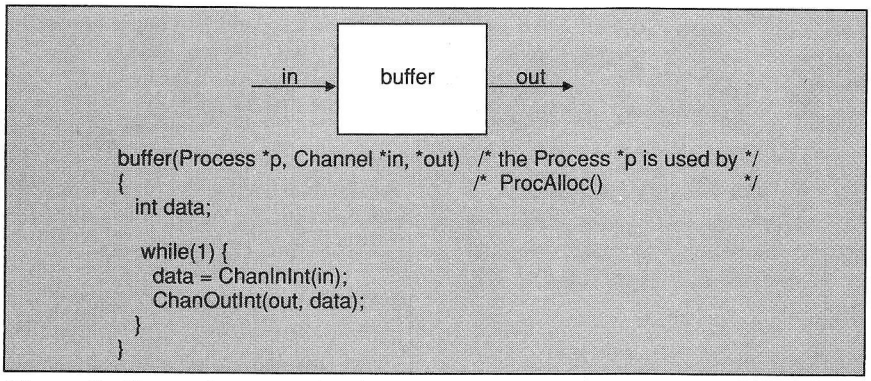
There are three pairs of functions for channel communication. *ChanIn( )*/*ChanOut( )* transfer an arbitrary number of bytes of data; *ChanInInt( )*/*ChanOutInt( )* transfer a single integer; and *ChanInChar( )*/*ChanOutChar( )* transfer a character. There are also channel routines for fault-tolerant applications that abort after a user-specified timeout period or by communication on another channel.

The alternative functions operate on lists of channels. *ProcAlt( )* and *ProcAltList( )* scan a list for inputs. The program blocks (waits) until one channel is ready for an input and then selects a channel. *ProcSkipAlt( )* and *ProcSkipAltList( )* scan a list of channels; however, process blocking is not performed.

Ken is a graduate student working on parallel processing and controls applications in the Smart Materials and Structures Laboratory at Virginia Tech. He can be contacted at the Mechanical Engineering Department, Randolph Hall, Virginia Tech, Blacksburg, VA 24061.

Channel Communication	Channel Status	Concurrency (CSP Model)
ChanAlloc ChanFree ChanIn ChanInChanFail ChanInChar ChanInInt ChanInTimeFail ChanOut ChanOutChanFail ChanOutChar ChanOutInt ChanOutTimeFail ChanReset	ProcAlt ProcAltList ProcSkipAlt ProcSkipAltList ProcTimerList ProcTimerAltList	ProcAlloc ProcFree ProcInit ProcPar ProcParam ProcParList ProcPriPar ProcRun ProcRunHigh ProcRunLow ProcStop ProcToHigh ProcToLow ProcWait
Concurrency (Fork/Join)	**Timing and Scheduling**	**Miscellaneous**
PFork PForkHigh PForkInit PForkLow PHalt PJoin PRun PSetup PStop	GetHiPriQ GetLoPriQ ProcAfter ProcCall ProcGetPriority ProcReschedule SetHiPriQ SetLoPriQ SetTime Time	BitCnt BitRevNBits BitRevWord Move2D Move2DNonZero Move2DZero restorefp _boot_chan_in _boot_chan_out _node_number

Table 1: *Partial list of LSC functions*




```
buffer(Process *p, Channel *in, *out)  /* the Process *p is used by */
{                                      /*  ProcAlloc()              */
  int data;

  while(1) {
    data = ChanInInt(in);
    ChanOutInt(out, data);
  }
}
```

Figure 1: *Channel communications*




```
mux(Process *p, Channel *in1, *in2, *out)
{
  int data, index;

  while(1) {
    index = ProcAlt(in1, in2, NULL);  /* return offset into chan list */
    switch(index) {
      case 0:                         /* channel in1 */
        data = ChanInInt(in1);
        ChanOutInt(out, data);
        break;
      case 1:                         /* channel in2 */
        data = ChanInInt(in2);
        ChanOutInt(out, data);
        break;
    }
  }
}
```

Figure 2: *Multiplexer using the channel alternative*

(continued from page 70)
If no channels are ready for input, the function returns immediately. *ProcTimerAlt( )* and *ProcTimerAltList( )* block the current process until one of the channels is ready or until a user-specified time elapses. An example of the alternative process is shown in Figure 2.

I like the use of C language extensions for readability, but I think libraries allow more flexibility for different parallel architectures

There are several methods for running an arbitrary number of processes in parallel on a single chip. In any case, each process must be allocated — a stack frame must be built for each process and memory created for a process structure. *ProcAlloc( )* performs these tasks by returning a pointer to an initialized process structure. When the allocated processes are no longer needed, *ProcFree( )* returns the allocated memory to the heap.

Once all processes have been allocated, one of the process creation functions — *ProcRun( )*, *ProcRunHigh( )*, *ProcRunLow( )*, *ProcPar( )*, *ProcParList( )*, and *ProcPriPar( )* — can be used. These functions take process pointers and spawn time-sliced processes according to the function used. Code for two processes is shown in Figure 3.

Concurrent Sorting

To examine the routines described above, I have developed a simple ASCII sort of a string of characters. The sort is implemented on a pipeline of parallel processes and is a modified version of the sorting example in the Inmos *Transputer Development System* manual. Single- and multiprocessor implementations of the sorting algorithm are shown in Listings Five and Six, respectively. The two programs utilize the following files (only the formats of the main programs differ): Listing One (page 124) lists proto.h, Listing Two (page 124) mytypes.h, Listing Three (page 124) multisort.nif, and Listing Four (page 124) buffers.c.

(continued from page 72)

The single-processor implementation of the sorting routine (see sort.c, Listing Five, page 124) performs the actions in Example 1. In the schematic representation of this sort shown in Figure 4, the circles represent parallel processes and the lines represent the communication channels. Note that the data flows through the pipeline in sorted order. As a result, the further down the pipeline a process (or processor) is, the lower its utilization. This is acceptable for a single-processor implementation, but it is a waste of resources in a multiprocessor implementation because most of the processors would be idle most of the time.

Also note that the channel inputs and outputs match. For each byte (or word or array) output, data of the exact same length must be input. If the length of the data transfers does not match, the data transfer will deadlock. Since deadlock is probably the most common programming bug that occurs when using transputers, extra care must be taken to ensure that channel I/O matches. In addition, channel communication must be decoupled sufficiently to ensure that processes are not waiting for each other to communicate first.

The two-processor version (see multisort.c, Listing Six, page 126) performs the exact same sort, but the organization is slightly different. In this case,

Transputer Architecture

A transputer is a microprocessor developed by Inmos and designed specifically for Multiple Instruction Multiple Data (MIMD) parallel-processing applications. All transputers have on-chip memory and serial links for connecting one transputer to another. Transputers come in several flavors: 16-bit, 16-bit with disk control logic, 32-bit, and 32-bit with an on-chip floating-point unit. The current generation of transputers has 4 Kbytes of on-chip RAM and four bidirectional serial links that can operate at 20, 10, or 5 Mbits/sec.

In addition, transputers have a microcoded scheduler with two priority levels, which allows concurrent (time-sliced) processes to share a processor's time. There are also two sets of timers, one for each priority level. A block diagram of the Inmos T800 floating-point transputer is shown in Figure 7. A T800 transputer offers approximately the same performance as the Intel 80386/80387 chip combination at around half the cost because you buy one chip instead of two.

Transputer Concurrency Model

The transputer concurrency model is based on C.A.R. Hoare's "Communicating Sequential Processes" (CSP) (Hoare, 1978). The "native" transputer language Occam (named after the 14th-century philosopher William of Occam) is based on the CSP idea of concurrency and communication (May and Taylor, 1984). A process performs a sequence of actions and then terminates. Concurrent processes can be thought of as multiple black boxes, each of which has its own internal state. Communication between processes is performed through point-to-point channels (CHAN). Each channel provides a one-way connection between two processes. It is not possible to share a channel between more than two processes, nor is there shared memory between processes. Information is updated through channel communication, thus alleviating the need for memory locks such as those used on shared memory multiprocessors. Communication between processes is synchronous and unbuffered. One process is blocked (descheduled) until the other is ready to communicate. When both the sending and receiving processes are ready, they proceed.

The communication channels can be virtual channels maintained internally by a single transputer, or they can be mapped onto one of the hardware links for true parallel processing applications. The transparent nature of virtual and physical channel/link communication enables the programmer to develop many programs on a single transputer and, with minor modification, map the program onto a network of processors.

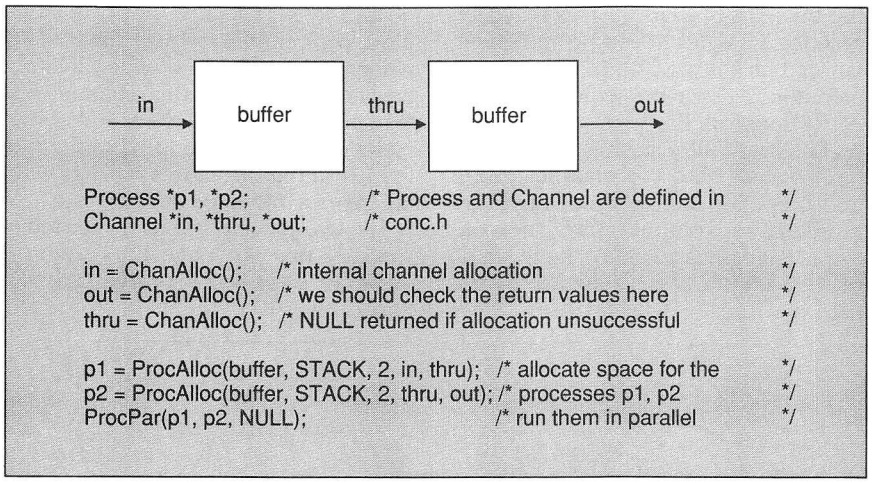
Another channel construct is the alternative. The alternative (ALT) construct provides a method of choosing the first available channel from a set of channels and is implemented on the input side of channel communication. ALT is often used to control the flow of data through a network.

To generate concurrent (or virtual concurrent) processes, a parallel construct (PAR) is used. As previously mentioned, parallel processes can only communicate through channels. The transputer supports high priority (used for communication processes) and low priority (for computation processes). While this may seem backward, it is important in a multiprocessor environment to keep as many processors as busy as possible by providing them data to process. Low-priority processes get time-sliced approximately every 1 millisecond. High-priority processes do not get time-sliced. They will, however, deschedule while waiting for channel communication.

Transputer Environments

Transputer networks generally consist of a root transputer that interfaces directly with a host such as a PC, Macintosh, or Sun. All other transputers in the network are connected in some fashion to the root transputer using the built-in serial links. Even though these network nodes may reside in the host, they rely on the host only for power and ground. Only the root transputer communicates with the host computer.

Typically, the root transputer is connected to an adapter chip that converts a transputer serial link to a bus interface. The host runs some sort of file server program to service the transputers. In the case of the C compiler, the server provides access to the keyboard, monitor, and disk storage of the host. The LSC compiler has the hooks in its library so that the standard C library functions communicate correctly with the host server. However, at this point in time, only the root transputer can use *stdio* routines to the host without extra programming effort. — G.K.E.




```
Process *p1, *p2;              /* Process and Channel are defined in      */
Channel *in, *thru, *out;      /* conc.h                                  */

in = ChanAlloc();    /* internal channel allocation                       */
out = ChanAlloc();   /* we should check the return values here            */
thru = ChanAlloc();  /* NULL returned if allocation unsuccessful          */

p1 = ProcAlloc(buffer, STACK, 2, in, thru);  /* allocate space for the    */
p2 = ProcAlloc(buffer, STACK, 2, thru, out); /* processes p1, p2          */
ProcPar(p1, p2, NULL);                       /* run them in parallel      */
```

Figure 3: *Two parallel processes code fragment*

```
Root Transputer:

SEQUENTIAL
   - Read in line of text and store in character buffer.
   - Allocate channels and processes.
   PARALLEL
       - Send out a character at a time to the first pipe process.
       - In 256 parallel pipe processes: read in a character then
           read a new character, forward the character with the
           lowest ASCII value.
       - Read in one character at a time from last pipe buffer
         and write it to the screen.
   - Deallocate channels and processes.
```

Example 1: *Actions performed by the single-processor implementation of the sorting routine.*

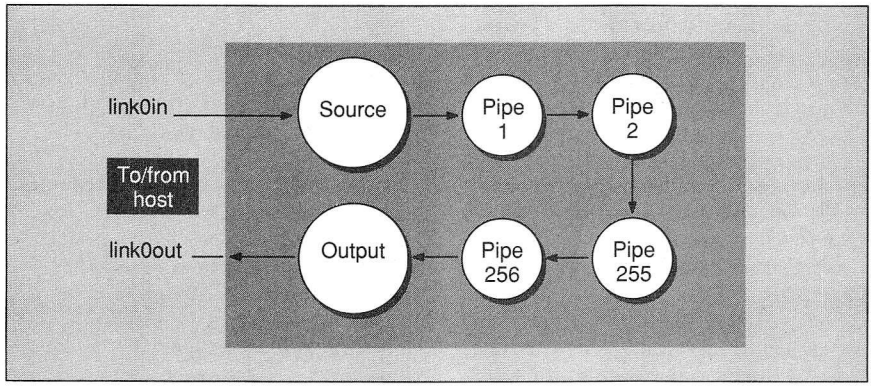


Figure 4: *Single-processor sort*

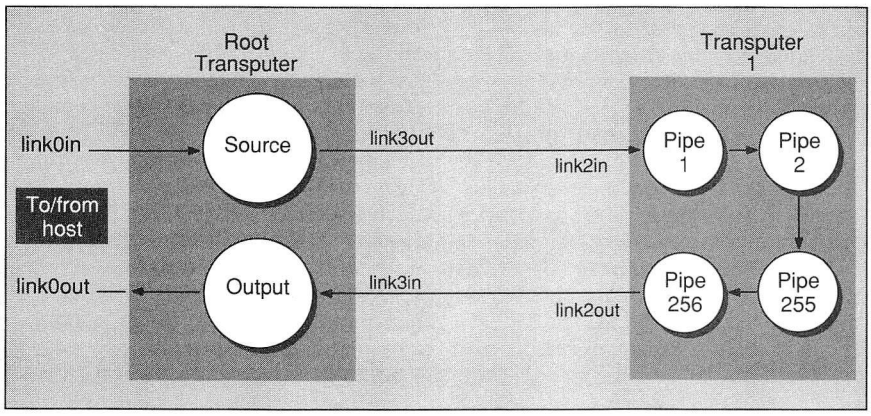


Figure 5: *Two-transputer sort*

(continued from page 74)

the root processor merely reads the character string, then forwards it out a hardware link to the next processor.

At the same time, the sorted data coming from the network is read in and displayed a character at a time. The block diagram of this is shown in Figure 5. In this case, a rectangle represents a transputer and the circles represent concurrent processes (see Example 2). A method for mapping the program from two processors onto a larger transputer network is illustrated in Figure 6. The only differences in the two-processor version are that the pipe processes are distributed among more processors, and some of the channels are mapped onto hardware links instead of using internally allocated channels.

I have written the sorting program in a somewhat unusual manner: The code for each processor of the multitransputer version of the sort is identical

I've written the sorting program in a somewhat unusual manner: The code for each processor of the multitransputer version of the sort is identical. Normally, I would write a unique root transputer program and then another program to be used for all of the additional network nodes. However, because I have only two processors, I've set up the program to determine on which node the code is running and then perform the appropriate actions. The *_node_number* variable is supplied to each node by the network loader.

To modify the code for more processors, forward the data to another processor with more sorting processes instead of having Processor 1 above return the data to the host. The last processor in the network can then be connected back to one of the other root transputer links.

The sorting algorithm uses the *source( )*, *output( )*, and *pipe( )* functions, which are the same for every version of the program. Only the for-

mat of the main program varies from the single- to multitransputer case.

Functions run in parallel require a Process pointer parameter as the first function in the parameter list. This pointer is used by the LSC process allocation function. Aside from its appearance as the first variable in the parameter list, its use is completely transparent to the programmer.

- *source( )* takes as its parameters a pointer to an output channel and a pointer to the character buffer where the text to be sorted is stored. A single character at a time is output down the specified channel. When an ASCII NULL is encountered, the function exits.
- *pipe( )* takes pointers to both an input and an output channel. Two characters are read in, and the one with the lowest ASCII value is forwarded to the next pipe process. Then a new character is read, a comparison is performed, and the character with the lowest ASCII value is forwarded. When a NULL is received, the stored character is forwarded, the NULL is forwarded, and the function exits.
- *output( )* reads a character at a time from an input channel and displays it using the *putchar( )* function. The function exits when a NULL is received.

In the single-processor case, each time a line of characters is sorted, the channels and processes are allocated, used, and then deallocated. For multiprocessors, only the root node allocates and deallocates each time through the loop. The network pipe buffer processes are allocated once, and terminate only as the root processor exits to the host. Actually, the network nodes deadlock due to lack of data from the root node, but this is okay as long as the root node shuts down gracefully. This implementation is used to keep the network programs as simple as possible and to circumvent the lack of support for multiple processes on the root to being able to communicate using the *stdio* stream.

Conclusion

This article provides a basic idea of how you can take advantage of the features available on transputers for building parallel processing networks. The ease with which parallel networks of transputers can be programmed and expanded, and the performance achieved by using this method, enable programmers to use transputers to solve problems that were previously relegated to large and expensive computing resources.

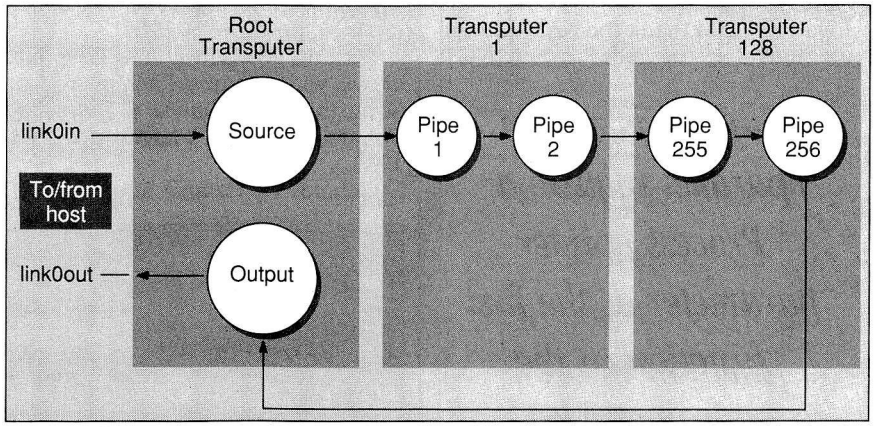


Figure 6: *128-transputer sort*

Acknowledgements
I would like to thank the NASA Graduate Student Researcher's Program (NGT-50392) for their support, and Kirk Bailey at Logical Systems for answering my questions about the compiler.

Functions run in parallel require a Process pointer parameter as the first function in the parameter list

References

Hoare, C.A.R., "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21, No. 8 (August), pp. 666 – 677, 1978.

May, David and Taylor, Richard, "Occam — an Overview," *Microprocessors and Microsystems*, Vol. 8, no. 2 (March), pp. 73 – 79, 1984.

Stevens, Al, "From C to C++: Interviews with Dennis Ritchie & Bjarne Stroustrup," *Dr. Dobb's C Sourcebook*, Vol. 14, no. 159 (Winter), pp. 9 – 17, 1989.

Gehani, N. H. and Roome, W. D., *Concurrent C, Computer Technology Research Laboratory Technical Reports*, Concurrent C Project, AT&T Bell Laboratories, Murray Hill, NJ 07974.

Henderson, Brian, "Par.c System — a C Compiler for Transputers," *Mailshot April 1990, SERC/DTI Transputer Initiative*, Rutherford Appleton Labora-

```
Root Transputer:

SEQUENTIAL
    - Read in line of text and store in character buffer.
    - Allocate channels and processes.
    PARALLEL
        - Send out a character at a time to the first buffer.
        - Read in a character at a time and write it to the screen.
    - Deallocate channels and processes.

Transputer Node 1:

SEQUENTIAL
    - Allocate channels and processes.
    PARALLEL
        - Read in data from link a character at a time, sort, forward to
          next pipe process.
        - In 254 parallel pipe processes: read in a character then
          read a new character, forward the character with the
          lowest ASCII value.
        - Read in data from pipe process sort, forward data to
          root transputer.
```

Example 2: *Actions performed by the two-processor implementation of the sorting routine.*

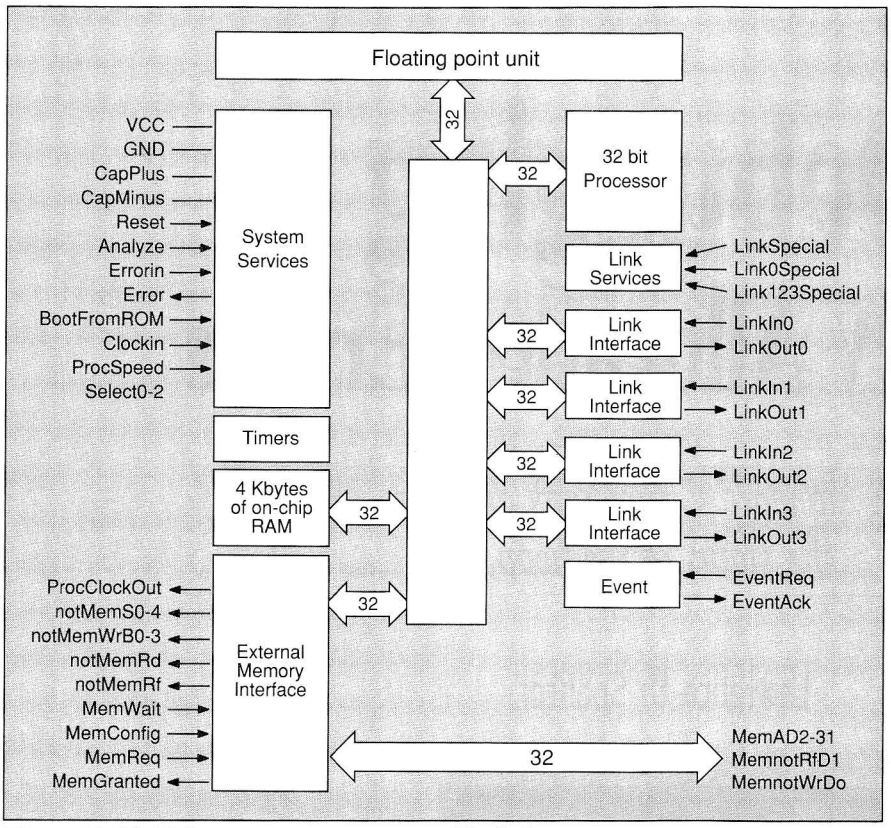


Figure 7: *T800 floating-point transputer block diagram*

tory, Chilton, DIDCOT, Oxon, OX11 0QX, pp. 45 – 51.

Logical Systems Transputer Toolset (Version 89.1), C Library Description, Logical Systems, P.O. Box 1702, Corvalis, OR 97339.

Transputer Development System, INMOS, Ltd., Prentice-Hall, New York, 1988.

Source Code Availability

As a service to our readers, all source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Drive, Redwood City, CA 94063, or call 800-356-2002 (inside Calif.) or 800-533-4372 (outside Calif.). Specify issue number and disk format. Code is also available through the *DDJ* listing service (415-364-8315) and through the DDJ Forum on CompuServe (type GO DDJ).

DDJ

(Listings begin on page 124.)

Vote for your favorite feature/article.
Circle Reader Service **No. 6.**

Debugging Memory Allocation Errors

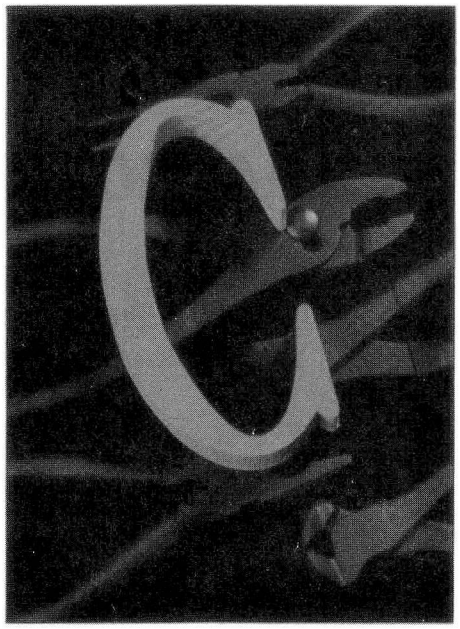
Replacing standard C functions and checking the status of the heap

Lawrence D. Spencer

It's 10 a.m., and you are working on your new real estate analysis program, HIGHRISE, that's scheduled to be finished tomorrow. The high-powered analysts will soon be sitting in their high-backed, leather chairs, clicking on your high-rise icon, eager to find out which property will drive profits still higher. However, you're feeling pretty low. HIGHRISE works great, but once in a while it locks up the computer. Usually, this happens after about a half hour of operation. Also, what about the time HIGHRISE forecasted a profit of $17,546,321.97 on a $1,000 investment? You never could reproduce the problem, so it was probably a fluke, right? "Please, let it have been a hardware error!"

Of course, if you know enough C to be interested in this article, you know that there is a 99 percent chance that the problem is related to memory allocation. Maybe you freed a pointer that you never allocated. Much like trying to walk down stairs that have not been built yet. Maybe you allocated memory but never freed it. A slower, agonizing death by asphyxiation.

You are probably thinking, "If I know enough C to be interested in this article, I just don't make that kind of mistake." Don't be so sure. About half the commercial database and screen-handling libraries I have worked with fail to free all the memory they allocate, even after their shutdown functions have been called! If the pros can make this mistake, so can we all.



The only way to find HIGHRISE's problem is to identify each memory allocation, make sure there is a corresponding *free( )*, then find each *free( )* and make sure there is an allocation for it. But HIGHRISE has now grown to half a meg of source. Even with a sophisticated debugger, you face a formidable task.

I'll show you an easier way. In the first part of this article, I'll present some functions that can replace the standard functions *malloc( )*, *free( )*, and so on. Each replacement function tracks your program's activity and reports it in a debugging file. If all is well, it calls the standard library function to do the real work. Using these replacement functions, I've tracked down some very cunning bugs, and you will, too. In the second part, I'll show how to obtain the heap-status of the heap without any special programming. A few well-placed calls to the functions may help you home in on the section of code that is giving you trouble. This technique described in the second part turned up the errors in those commercial libraries mentioned earlier.

Replacement Functions

Look at the program in Listing One (page 178). This program is a disaster. One block of memory is allocated but never freed. If this happens enough times in a program, it runs slower and slower, until it finally runs out of memory and hangs. Another block in Listing One is freed without ever having been allocated. In DOS, that kind of mistake can even clobber your file allocation table. (I did once!) Either type of error can be almost impossible to track down in a large, complex program.

Now look at Listing Two (page 178). I have replaced the calls to the stan-

Larry is president of Cornerstone Systems Group, a firm that specializes in C and C++ consulting and training. Larry is also the secretary of the Connecticut chapter of the Independent Computer Consultants Assoc. He can be reached at 10 N. Main St., West Hartford, CT 06107; 203-236-9209.

dard functions with calls to the replacement functions that I will now describe.

First, notice the calls to *memMalloc( )*. *memMalloc( )* works just like *malloc( )*, but there is a second argument. This argument is a "tag" that you can use to identify the call in the debugging file. This is the pattern for all the replacement functions: You use them just like the standard functions, but they take a tag as an extra argument.

The calls to *memFree( )* replace calls to *free( )*. Again, a tag is added to the argument list. Listing Three (page 178) shows the replacement routines. We'll begin with *memMalloc( )*. This function begins by calling *malloc( )*. *memTrack_alloc( )* is then used to track the allocation.

memFree( ) calls *memTrack_free( )*, which tracks the attempt to free, and returns TRUE if the memory may in fact be freed or FALSE if not. Only after determining that the memory may be freed does *memFree( )* free it. The *memTrack* routines are obviously the key to the whole scheme, so let's discuss them next.

memTrack_alloc( ) and *memTrack_free( )* keep track of pointers allocated and freed. The implementation shown here keeps a journal of activity in a file, but you could easily write an implementation that keeps records in an array or a binary tree. Although the file method is slower, it has the advantage of surviving the crash that you are trying to debug!

As you can see in *memTrack_alloc( )*, the file consists of records of three fields each. The first field is either an A (for allocated) or F (for freed). The second is the address of the allocated item. The third is the tag you supplied in your call to the routine.

memTrack_alloc( ) simply appends a record of the allocation that it is tracking. Part of this record is the allocated address, in %p (pointer) format. This formats the address as 0000:0000. The %p format is not available with some compilers, so you may want to use a long integer, or whatever is appropriate on your machine.

memTrack_free( ) searches through the file, looking for confirmation that the memory you are about to free was, in fact, allocated. If it finds an "allocated" record with the same memory address, it over-writes the A (allocated) with an F (freed). If it does not find such a record, it writes an error message to the file.

These routines call the function *memTrack_fp( )*, which returns a FILE pointer for the debugging file. The name of this file must be in the environment variable MEMTRACK. In DOS, this is accomplished with something like: *SET MEMTRACK=C: \ MYPROG.MEM*

You may want to use a RAM drive if your program is at least giving you a normal exit. (If the program locks up your system, you will have no way to get to the RAM drive!)

If MEMTRACK is not set in the environment, nothing will hang — you just won't get a debugging file. This provides an easy way to turn off memory tracking without recompiling all your code.

Finally, *memTrack_msg( )* appends a message to the debugging file. If, for some reason, the debugging file could not be opened, the message goes to standard error.

Speaking of the debugging file, check Figure 1 to see what it looks like now. Recall that the memory for "tag 1" was allocated but never freed. In the debugging file, this results in a line that starts with an "A" (for allocated). The memory for "tag 2" was allocated and

```
A 2524:000C tag 1
F 2524: 021A tag 2

Tried to free 113A:000B (tag 4). Not
allocated.
```

Figure 1: *The debugging file after* memTrack_msg( )

freed properly. *memTrack_free( )* has over-written an "A" that used to be on line 2 with an "F." This is how you want all your debugging lines to be. Finally, there is a real problem with the line for tag 4: We tried to free a pointer that we never allocated.

Of course, all this housekeeping takes extra processing time so we want to be able to use the standard functions alone in final production. The *#include* file mem.h (Listing Four, page 180) takes care of this for us. Notice the statement *#ifdef MEMTRACK.*

In a complicated program, the best approach is to call heapPrt( ) *in places where you can predict the status of the heap*

If you have *#defined* MEMTRACK, the header file emerges from the preprocessor as a set of function prototypes. If you have not (*#else*), the calls to the replacement routines are *#defined* to be calls to the functions in the standard library. Thus, you will incur no extra overhead in your tested, production version. Even the tag strings will not take up static space in your executable program; they vanish at preprocessing time, when the *#define* is processed! You can cause MEMTRACK to be *#defined* by coding *#define MEMTRACK* in each of your C programs, ahead of your *#include* mem.h. Alternatively, you can use a command-line switch on your compiler. With the Microsoft compiler, for example, type *cl /c /DMEMTRACK program.c.*

I prefer the command-line method because it is easier to undo. I just recompile without the switch. The in-source method does, however, have one advantage: You can tell whether a given module was compiled with MEMTRACK simply by looking at the source.

If you're careful, you can even *#define* MEMTRACK selectively. That is, you can compile one subset of programs with /DMEMTRACK and another without. For this to work, each subset must be self-contained as far as memory allocation is concerned — neither one should free memory allocated by the other.

So far, we have replaced only *malloc( )* and *free( )*. However, there are several other routines in the standard library that allocate memory. We must replace all of them, or we will not have a complete record of our activity. For example, if we do not replace *realloc( )*, our debugging file will have no record of its allocation. *memFree( )* will then raise a spurious objection when we try to free the memory. The other functions in Listing Three complete the job by replacing *calloc( )*, *realloc( )*, and, don't forget, *strdup( )*.

To debug a program with the replacement functions, then, there are five steps:

1. #include <mem.h>
2. Use the *mem* routines instead of the standard ones
3. Compile with MEMTRACK *#defined*, either in the source- or on the compiler's command line
4. Link with the *mem* routines
5. Set the environment variable MEMTRACK equal to the name of your debugging file.

To turn off debugging you have two choices:

- Recompile everything without MEMTRACK *#defined*, and link without the replacement routines. This is the preferred method for a commercial release.
- Take MEMTRACK out of your environment. The replacement routines will still be a part of your code, but the *memTrack* routines will not do anything.

Heap Status

Sometimes the situation is so awful, and your program so large, that it is impractical to begin with the approach presented earlier. Or you may think the situation is perfect, and you just want to be sure. The routines that follow give you a summary of the heap's status with one function call. These routines are built on some that Microsoft provides with its C compiler. In the hope that you use Microsoft, or that your compiler provides similar routines, I now refer you to Listing Five (page 180).

You will see that I have made all the same mistakes I made in Listing One. But this time, I have interspersed calls to the *heapPrt( )* function. This function tells me how many allocations I have made, and how many bytes they total. By default, *heapPrt( )* writes to standard output, with *puts( )*. Depending on what else your program does to the screen, this may not be what you

want. I have, therefore, coded *heapPrt( )* so you can designate your own message function. In Listing Five, we use *my_own_msg_func( )*.

The output at the end of the listing tells the story: We have 10 bytes yet to be freed when the program is done. This is never good, if only from a stylistic point of view. Now, some programmers like to let the operating system free the memory when the program exits. If you are one of those programmers, I tell you that this is like letting your wife or mother pick up your dirty laundry. Sure she'll do it, but sooner or later it's going to bite you. Maybe your program will become a subroutine and strangle your application after enough calls. Or maybe you will have a *real* memory problem that will be difficult to find amidst all the sloppiness.

Whatever your relationship with your mother, I think you can appreciate the usefulness of *heapPrt( )*. It takes a global view, while the *mem . . . ( )* routines in the first part of this article are more focused. *heapPrt( )* is particularly useful when you suspect that there is unfreed memory in someone else's code and you can't replace all his *malloc( )*s with the routines in this article.

In a complicated program, the best approach is to call *heapPrt( )* in places where you can predict the status of the heap. Often, you will predict that two successive calls to *heapPrt( )* will produce the same output, because the code in between is supposed to free all the memory it allocates. If your expectation is not met, you can zero in on the problem with more *heapPrt( )* calls, or use the *mem . . . ( )* routines.

The code for *heapPrt( )* and an associated function, *heapUsed( )*, is in Listing Six (page 180).

A Freebie

At the beginning of this article, I said that 99 percent of slow deaths are caused by memory allocation mistakes. What about the other one percent? Those are the programs that open files but never close them. Eventually, you run out of file handles. That problem is analogous to the memory allocation problem, and I have written a suite of functions to handle it. They are included in the file you can download from this magazine's bulletin-board service. You also may obtain the whole package free of charge by writing me at Cornerstone Systems Group, 10 N. Main St., West Hartford, CT 06107. Please include a formatted, IBM PC-compatible disk.

Summary

I have presented two sets of functions. The first can serve as a bookkeeping layer between your programs and the standard memory management functions. The bookkeeping can be turned off by a quick change to your environment. The bookkeeping layer can be removed completely by recompiling. The second set of functions will give you a snapshot of the heap at any point. Either set of functions can save hours of time in very difficult situations.

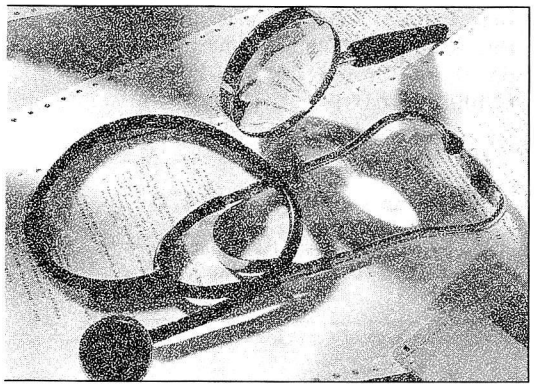
Source Code Availability

As a service to our readers, all source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Drive, Redwood City, CA 94063, or call 800-356-2002 (inside Calif.) or 800-533-4372 (outside Calif.). Specify issue number and disk format. Code is also available through the *DDJ* listing service (415-364-8315) and through the DDJ Forum on CompuServe (type GO DDJ).

DDJ

(Listings begin on page 178.)

Vote for your favorite feature/article.
Circle Reader Service **No. 7.**



Optimizing with Microsoft C 6.0

Based pointers and global optimization highlight features of this new incarnation

Scott Robert Ladd

For two years, Microsoft C 5.1 has been the benchmark by which other C compilers are judged. Vendors of competing compilers always compare themselves against Microsoft's product. Some competitors have edged ahead of Microsoft C 5.1 — Watcom C 7.0 produces faster programs; Borland's Turbo C and others are faster at producing programs. While Microsoft C 5.1 has remained at the top of the heap, it is no longer the leader in many categories.

The upgrade has finally arrived in the form of Microsoft C 6.0. Microsoft's answer to the competition includes a new programming environment, an improved code optimizer, a new version of the venerable CodeView debugger, and some new library enhancements. Pulling out all the stops, Microsoft has made clear its intent to keep its C compiler at the forefront of the industry.

An Overview

Perhaps the biggest surprise for many is that Microsoft C 6.0 (MSC6) does not include any C++ extensions. With most other vendors jumping on the C++ bandwagon, it might seem odd that Microsoft has not followed the same path. Some industry watchers have opined that Microsoft has made a mistake by leaving out C++. I don't agree — C++ is a language that is still maturing. Most C programmers are sticking with C; only a few have embraced C++ as their primary programming language. For now, C is the dominant programming language for professional developers of PC applications, and Microsoft is addressing this market.

You must use Microsoft's Setup program to install MSC6. The files on the distribution disks are compressed, and they can be expanded only by using the Setup program. Setup is a very good installation program; it gives you the choice of installing different sets of libraries and tools based on your needs. Additional libraries can be built later without having to reinstall the entire package.

MSC6 is disk-hungry; installing the MS-DOS version of the compiler, the programming environment, two memory models (small and large) each for the floating-point emulator and the coprocessor, and other tools used over 7 Mbytes of space on my hard drive. If you install all four memory models for all three floating-point options along with the OS/2 version of the compiler, you can easily use more than 12 Mbytes of disk space. Obviously, it is not possible to use MSC6 on a floppy disk-based computer!

One last change in the basic package may take some people by surprise: It is now distributed on 1.2-Mbyte 5 1/4-inch disks and 720K 3 1/2-inch disks. If you want MSC6 on 360K disks, you'll have to special order them after you've bought the package. Microsoft is offering MSC6 on CD-ROM, preinstalled and ready to run.

Documentation?

No, the question mark is not a typo. Looking at MSC6 you may wonder if Microsoft forgot to pack the documentation. Microsoft has minimized the paper documentation in their language products in recent years. Instead of providing thick books, Microsoft puts the majority of the documentation into on-line databases. MSC6 comes with only 940 pages of documentation, less than half the length of the manuals provided with Microsoft C 5.1 (MSC5.1). Going from the fat three-ring binders of MSC5.1 to the thin paperback manuals of MSC6 gives the impression that Microsoft has left something out.

The documentation isn't missing; it's just not where you'd expect it to be. The on-line help system is where the real documentation for MSC6 lies. There are nearly 2 Mbytes of help files covering every aspect of MSC6 in great detail. These help files are available either through the QuickHelp program, or from within the Programmer's Workbench environment.

In MSC5.1, the command-line compiler had a */HELP* switch that displayed a simple list of compiler switches. The same switch in MSC6 is now available for virtually every program in the package. Instead of a simple command list,

Scott is a full-time freelance computer journalist. You can reach him at 705 West Virginia, Gunnison, CO 81230.

though, */HELP* will invoke the interactive menu-driven QuickHelp system. This provides complete information on the program in question. Entering the command *lib /HELP* will bring up detailed information on the library manager utility. The scope of the information runs from an explanation of *lib* to complete coverage of *lib* switches and commands (with examples).

The lion's share of the help information is available only when you are working within the Programmer's Workbench. If you're working with OS/2, the QuickHelp utility can be used as a keyboard monitor, so that it can be popped up at any time for use. Unfortunately, MS-DOS users do not have the luxury of a TSR version of QuickHelp.

Considering the amount of on-line documentation, I was pleased to find that the seemingly small amount of paper documentation was actually very useful. There are two paperback manuals in the MSC6 box: the spiral-bound 380-page *Microsoft C Reference* and the 480-page *Advanced Programming Techniques*. The amazingly long title *Installing and Using the Professional Development System* belongs to an 80-page booklet that explains the installation process and the fundamentals of using the Programmer's Workbench environment.

The *Microsoft C Reference* begins with a section describing each program provided by Microsoft. For the compiler and utilities, a list of command-line parameters and switches is given. For *nmake* — Microsoft's much-awaited enhanced *make* utility — a reference to

Based Pointers for Optimization

Bruce D. Schatzman

Microsoft C 6.0 provides an important new tool in the battle to keep code small and fast — the based pointer. Virtually all Microsoft C programmers are familiar with *near* and *far* pointers, and how they are used within standard or mixed memory models. The 2-byte *near* pointers provide both size and speed advantages over the larger and slower 4-byte *far* pointers. As such, programmers tend to use *near* pointers within small memory models whenever possible.

Even the most carefully designed small model programs, however, sometimes hit the memory wall of DGROUP's 64K allocation. Programmers are then faced with a tough decision: Find a way to keep the program within the confines of a Small memory model to preserve the speed advantage, or admit defeat and move to larger and less efficient models such as *compact* or *large*. The inefficiency of larger models is primarily the result of the use of 4-byte (*far*) pointers. Each time a *far* data item is referenced, both the segment and offset address must be loaded into one of the 80x86's segment registers. This double load consumes more CPU cycles than *near* pointers (which load only an offset address), and increases both code and data sizes.

Microsoft C 6.0 provides a solution to this problem through the use of based pointers. This new data type gives you the reach of *far* pointers while retaining the size and speed advantages of *near* pointers. You can now increase your program's data space without necessarily moving to a larger memory model or a mixed (*near/far*) model.

Listing Five (page 131) presents a small skeleton program that sets up a linked list of file names using a set of structures (*TAG*), with a maximum number of *MAX_TAG* structures. Two new keywords are used: *_segment* and *_based*. The statement *_segment segvar;* declares a variable that will hold a segment's memory address. *segvar* will become a segment in which a set of data is based — thus the name "based pointers."

This basing is illustrated in the statement *_based(segvar) *PTAG, TAG;*, which defines a 2-byte structure pointer and a structure that are both based within the segment *segvar*.

The compiler generates code that automatically looks at the *segvar* variable each time a reference is made to **PTAG* and *TAG*, necessitating only the use of a 2-byte offset address. Technically, **PTAG* and *TAG* are both in a *far* heap, yet they act as if they reside in a *near* heap.

If *MAX_TAG* were smaller, this program could fit comfortably within a small memory model. However, with *MAX_TAG = 2000*, the tag list itself could fill up a 64-Kbyte data segment. Basing our structures within *segvar* means that as the program runs, successive references to these structures involve loading only the 2-byte segment offset. Because the structures are all within one segment, the segment address is preloaded into one of the segment registers and does not need to be reloaded every time a new *segvar* structure is referenced.

However, referencing another named segment (or *far* pointer) will requires loading a new value within a segment register. Therefore, based pointers are most effective for use with successive references to data items within the same segment, and offer performance equivalent to that of *near* pointers.

Note the program's use of the new C 6.0 library routines *_bheapseg* and *_bmalloc*, which allocate a based-heap segment and memory within that segment.

Although I've focused here on keeping small programs small, based pointers can also be used to reduce the size (and increase the speed) of large programs. This is accomplished by converting some or all of a large program's *far* pointers to based pointers.

Based pointers are an important tool, but they have their limitations. Perhaps the most significant is the fact that their actual addressing capability is 16 bits — the size of a single segment. They do not have the unlimited 32-bit scope of *far* pointers, and thus cannot be used for very large data structures, such as arrays that exceed 64K. *far* pointers are the only real choice in these circumstances. Nevertheless, based pointers are a tool that will undoubtedly be used by a large percentage of Microsoft C programmers to optimize code for smaller size and greater speed.

Bruce Schatzman has worked in the computer industry for over 10 years, holding a variety of technical and marketing positions at corporations including General Dynamics, Tektronix, and Xerox. He is currently an independent consultant in Bellevue, Wash., specializing in systems consulting and technical communications.

makefile statements is provided. The section on CodeView presents a complete list of all debugger commands and data formats. The documentation for Programmer's WorkBench includes tables explaining the environment's numerous switches, macros, functions, and key assignments.

Following the utilities section, the *Microsoft C Reference* has a complete list of all library functions, providing information on prototypes, required header files, return values, and compatibility between operating systems. Each function has a one- or two-sentence description, which is adequate for an experienced C programmer. The appendices list *printf* and *scanf* format specifiers, key codes, and ASCII values. I think the *Microsoft C Reference* is the longest quick-reference manual I've seen, but it has quickly become indispensable.

A wide variety of topics are covered in *Advanced Programming Techniques.* It contains chapters on optimization, floating-point operations, memory management, the new in-line assembler, project development, debugging, graphics, and mixed-language programming. This how-to book provides details on how the MSC6 system can be used most effectively.

My overall impression of the documentation is that it was designed for an experienced professional. I would not recommend this product to someone just beginning to work with C. Beginners need hand-holding and detailed documentation, something that Microsoft is not providing with MSC6.

The Compiler and Tools

QuickC is not included with MSC6. However, the core compiler for QuickC is there; it can be invoked by the compiler control program with the */qc* switch. Using */qc* doubles the speed of compiles, but prevents you from using advanced optimizations. The compiler itself has actually changed in a number of areas from MSC5.1 to MSC6. The primary additions are in improved ANSI compatibility, a Tiny memory model, an in-line assembler, optimizations, "based" pointers (see sidebar), and a new *long double* type.

MSC5.1 stuck closely to the ANSI standard; MSC6 is just about on the money. New ANSI features in MSC6 include complete support for *volatile*, *long* values in *switch* statements, and support for "locales." A locale describes the numeric, money, date, and time formats for a given country. As with most MS-DOS C compilers, MSC6 only supports the C locale, which is identical to the locale for the United States, and is the only locale an ANSI-standard compiler must support.

Microsoft has finally recognized that some programmers want to produce programs in the simple .COM format. The new Tiny memory model is a modified version of the Small model; the primary difference is in how a program is linked. Link combines a special library with Tiny model programs to directly create a .COM file without a call to *exe2bin*.

The in-line assembler was introduced by Microsoft in QuickC 2.01, and it's now available in MSC6. The *_asm* keyword can preface a block of assembler instructions in a C program. The *_emit* function places a series of arbitrary byte values into program code. While the in-line assembler and *_emit* are nonportable, they do provide a convenience many programmers will find difficult to resist (see "DOS + 386 = 4 Gigabytes," *DDJ*, July 1990).

As predicted, Microsoft has increased the power of its optimizer. Many new optimization options have been added, including global optimizations that look at functions as a whole when analyzing a program. MSC6 is now in the same class as Watcom and Zortech when it comes to optimizations.

MSC5.1 was known for the infamous aliasing problem. An alias occurs in a

(continued from page 86)
C program when two different names refer to the same memory location. An optimizer can produce faster code if it can assume that no aliases exist. Under MSC5.1, the maximum optimization compiler switch (*/Ox*) told the optimizer to ignore the existence of aliases. Many programmers complained when MSC5.1 generated non-functional programs from code that contained aliases. Microsoft has changed */Ox* in MSC6 so that it assumes aliases exist. You can still tell the optimizer to ignore the possibility of aliasing, but you must do so explicitly with the */Oa* switch.

A new feature of MSC6 is that optimizations can be turned on and off within a source file through the use of pragmas. This allows you to explicitly protect a piece of code from strong optimizations, for example. MSC6 can enregister parameters, an optimization pioneered by Watcom C. Individual functions or complete source modules can be compiled with enregistered parameters. Normally the parameters to a function are pushed onto the stack before the function call is made. When enregistered parameters are in effect, parameters are stored in registers when the function call is made. This saves a considerable amount of pushing and popping of parameters on the stack, and can increase the speed of a program that makes numerous calls to functions with small numbers of arguments.

nmake is Microsoft's professional *make* utility. The older *make* program included with previous versions of Microsoft C was abysmally weak. Unlike its predecessor, *nmake* is compatible with Unix.

Debugging

CodeView 3.0 has been awaited with great expectations. Earlier versions of Microsoft's pioneering debugger were very good but clumsy to use. Products such as Borland's Turbo Debugger were easier to use and more powerful. While CodeView 3.0 goes a long way towards answering its critics, it's still a few steps short of what programmers want.

Tiled windows are passé — more information can be easily displayed in layered windows. Unfortunately, CodeView 3.0 uses tiled windows. With Programmer's WorkBench and other full-screen applications using a layered window system, CodeView seems a bit out-of-step. I could not use CodeView 3.0 with more than four windows open on a 25-line screen.

I also found CodeView 3.0 to be frustrating. I want CodeView to always come up in VGA 50-line mode with use of the 80386 debugging registers. While commands for CodeView can be placed in the TOOLS.INI initialization file, none of these commands change the default command-line parameters. There isn't even a CV environment variable to which default switch settings can be assigned. The compiler and linker both support environment variables that contain command-line switches; why CodeView doesn't is a mystery to me.

CodeView 3.0 can be run in extended memory (leaving more room for programs in main memory), but it can only do so when the HIMEM.SYS driver is loaded. HIMEM.SYS is incompatible with any other 386 control program, which prevented me from using Qualitas' 386MAX program, for example, to manage high memory when HIMEM.SYS is loaded.

CodeView 3.0 does not support the Virtual Control Program Interface (VCPI) for 386 programs, a standard developed by Phar Lap that allows multiple 386 virtual memory programs to operate in concert. Microsoft informed me that a VCPI version of CodeView may be developed, but did not name a release date. Make no mistake: CodeView 3.0 is superior to previous versions of the debugger. It has several new windows, including a "locals" view, which shows all variables local to the currently executing function. There is

(continued from page 88)
more flexibility in how windows can be tiled, and the command-line interface has largely been replaced by a set of keystrokes and mouse selections. Overall, CodeView 3.0 is a very powerful debugger. With a little work, it could be the best MS-DOS debugger available.

Benchmarks

Benchmarks are infamous for being both controversial and subjective, yet they are one of the few empirical tools we have for comparing compilers.

Optimization has become one of primary ways in which vendors differentiate their products from the competition. An optimizing compiler performs an analysis on a program being compiled, generating a more efficient program. Optimizers can delete unused code and variables, improve register use, combine common subexpressions, precalculate loop invariants, and perform other tasks that improve program performance.

At best, optimizing a well-written program will improve its speed by as much as 25 percent. An optimizer will not replace inefficient algorithms with better ones. As the saying goes, "garbage in, garbage out." Most of the responsibility for a program's performance lies with the programmer. Improving algorithms and manually optimizing a program will often increase program speed by several orders of magnitude. The purpose of an optimizer is to make sure that the compiler is producing the fastest code possible from your source code.

In recent years, Watcom's C 7.0 has become the standard by which optimizing C compilers are judged. Watcom entered the market in 1988 with a compiler that produced very fast executable programs. To see how well Microsoft C 6.0 stands up to the competition, I have run the benchmarks for it, as well as the benchmark for Watcom C 7.0. I've also included benchmark results for Microsoft C 5.10. These results will show current users of Microsoft C how much improvement they can expect from the new version. Table 1 shows the results of these benchmark tests.

Four programs make up the benchmark suite. Dhrystone 2 is a standard industry benchmark designed to represent the "average" program. For this test, it was set to run 200,000 iterations.

DMath is a test of floating-point code generation of my own invention. DMath calculates the sines of the angles between 0 and 360 degrees using a simple series. A test of floating-point code generation, DMath contains only *double* data items, and it makes no library function calls.

XRef is a test of dynamic memory management and I/O speed. XRef is a filter program that creates a cross-reference of input from standard input. The cross-reference is displayed to standard output.

The last benchmark program is GrafTest, a program that exercises my low-level graphics library. GrafTest performs millions of function calls, and interfaces directly with video hardware.

All tests were run on a 20-MHz i386-based computer running MS-DOS 3.30. The computer was equipped with a 25ms hard drive, a 20-MHz 80387 coprocessor, and a 16-bit VGA video system.

Two compiles were done for the benchmark chart. One compile used the compiler and linker options I would use when generating a program with debug information in it. The other compile used full optimization, inline math coprocessor instructions, and 80286 code generation. Most compiles done during development are with full debugging information on. For MSC6, I ran two sets of benchmarks — one with enregistered parameters, and the other without. Compile times are actually a combination of compile and link times. Timings are an average for five compiles/runs. The compiler command lines used are shown in Figure 1.

Table 1 shows some clear trends.

Enregistered parameters in MSC6 did not provide a significant increase in program speed in the DMath or XRef tests. The biggest gain from enregistered parameters is seen in the GrafTest benchmark, which makes hundreds of thousands of calls to functions that take two or three *int* parameters. Because Dhrystone 2 lacks function prototypes, MSC6 was unable to employ enregistered parameters. While Microsoft C 6.0 produced the fastest debug compiles and the best run times in three out of four tests, it fell far behind in run-time speed on the DMath test. Watcom maintains its position as the best compiler for mathematically demanding applications.

The OPT benchmark program is shown in Listing One (page 128). Each compiler was directed to produce an assembly language listing of its output, based on a compile using the optimized compiler switches just shown. Listing Two (page 128) shows the output from Watcom C 7.0; Listing Three (page 129) shows the output from Microsoft C 5.1; Listing Four (page 129) shows the output from Microsoft C 6.0 without enregistered parameters; Listing Five (see page 131) shows the output from Microsoft C 6.0 with enregistered parameters. These listings should give you a feel for the quality of code generation supported by the test compilers.

At the Workbench

Usually, I don't care much for integrated programming environments — the ones I've worked with have limited capabilities locked into a simplistic editor. Working from the DOS prompt has always been faster and more powerful. That is, until now.

The Programmer's Workbench is actually an enhanced version of the Microsoft Editor. The Microsoft Editor was easily as powerful as other professional program editors. It allowed separately compiled "add-ons" to be linked with it at run time. The Programmer's Workbench takes the idea of add-ons a step further by using them to add support for compilers, help systems, and utilities to its menus and environments.

The Programmer's Workbench editor is powerful and fast, incorporating all of the features programmers expect from a professional editor — recordable and compilable macros, powerful block operations, multiple file editing, and mouse support. It also allows you to give editor commands using menus or function keys. I haven't felt restricted by the Programmer's Workbench editor, and that's something I cannot say for other integrated editors.

You can do all of your program development work from within the Programmer's Workbench. The environment integrates the compiler, make utility, linker, help system, and CodeView relatively seamlessly. It's easy to "live" within the Programmer's WorkBench, leaving it only when you're ready to finish up for the day.

Programmer's WorkBench includes menus that allow you to set options for compiles. Options can be set for C, Microsoft Macro Assembler (MASM), and the linker. Every option available from the command line can be selected

Watcom C 7.0 debug	: -2 -7 -Os -d2 -ms
Watcom C 7.0 opt.	: -2 -7 -Oail -s -ms
Microsoft C 5.1 debug	: /c /G2 /FPi87 /qc /Zi /Od /AS
Microsoft C 5.1 opt.	: /c /G2 /FPi87 /Ox /AS
Microsoft C 6.0 debug	: /c /G2 /FPi87 /qc /Zi /Od /AS
Microsoft C 6.0 opt.	: /c /G2 /FPi87 /Oxaz /AS

Figure 1: *The compiler command lines used in the benchmark tests. The* /Gr *switch was added to the Microsoft C 6.0 command lines when enregistered parameters were used*

Program & Test	Watcom 7.0	Microsoft C 5.1	Microsoft C 6.0 no /Gr	Microsoft C 6.0 /Gr
Dhrystone 2				
time: debug compile	24.24	13.13	11.65	---
time: opt. compile	25.15	20.48	32.26	---
time: execution	31.88	29.42	27.25	---
.EXE file size	14,082	19,078	20,334	---
DMath				
time: debug compile	11.04	6.50	5.87	6.22
time: opt. compile	10.29	8.46	12.21	12.03
time: execution	30.39	40.64	40.48	39.41
.EXE file size	6,534	9.572	12,868	12,820
XRef				
time: debug compile	13.74	7.98	7.20	7.31
time: opt. compile	13.70	11.60	16.83	16.98
time: execution	33.48	32.42	32.41	32.37
.EXE file size	7,125	9,067	7,639	7,607
GrafTest				
time: debug compile	19.99	11.03	13.35	12.91
time: opt. compile	26.05	22.09	49.05	50.47
time: execution	27.71	28.78	27.09	26.85
.EXE file size	4,230	5,603	5,845	5,781

Table 1: *Benchmark results for Microsoft C 5.1, 6.0, and Watcom C 7.0*

via the menus. Two sets of program construction options can be set, for debug and production compiles. Combined with a list of files that are part of the same program, the options are used to construct a make file for a project. When you build a project, the project's make file is then passed as a parameter to *nmake*. Compilation results appear in a window, and errors can be tracked from the compiler's output directly into your source code. It's a slick, powerful, and uninhibited environment for constructing software. Microsoft has told me that the interface to the Programmer's WorkBench will be fully and publicly documented so that third parties can integrate their products into WorkBench. With luck, your favorite editor or *make* utility will become a part of the environment.

Programmers WorkBench has one drawback that may limit its acceptance by many programmers: It does not run efficiently on anything less than a 386-or fast 286-based PC. The environment is simply too slow to be useful when run on a 10-MHz or slower 286-based computer, according to my correspondents.

Other problems in the Programmer's WorkBench have surfaced. It requires nearly 3 Mbytes of disk space, room many developers need for other software and data. If you don't install WorkBench and its help files, you no longer have access to the primary source of detailed documentation for MSC6. For these reasons many developers may opt to leave Programmer's WorkBench off their system.

Conclusion

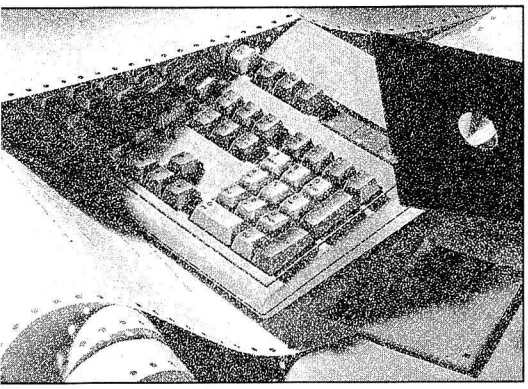
Microsoft has made a valiant effort at producing the definitive C compiler for MS-DOS and OS/2. For developers who have high-end PCs with available disk space, Microsoft C 6.0 is a solid professional product that offers some advantages over its competitors. Improvements in code optimization, compiler speed, and ANSI compatibility are major pluses.

Unfortunately, Microsoft missed the mark in some areas. CodeView 3.0 is not what programmers were expecting and the Programmer's WorkBench is a professional environment that is too bulky and too slow for many developers. Finally, the lack of complete paper documentation is simply inexcusable in a product priced at $450. Microsoft still has some work to do before they can accurately claim to have the best C compiler on the market.

DDJ

(Listings begin on page 128.)

Vote for your favorite feature/article.
Circle Reader Service **No. 8.**



Collections in Turbo C++

A full C++ 2.0 implementation makes writing collection classes a breeze

Bruce Eckel

According to the unwritten rules of implementing C++, compiler vendors have the option of saying "sorry, not implemented" when a language feature is either too difficult to implement or when the implementor feels that the programmer really doesn't need a particular feature. Most implementors have invoked this rule at one time or another, even AT&T with its cfront translator. Furthermore, vendors sometimes implement features for which a compiler may accept the syntax but not enforce it; *const* and *volatile* member functions are good examples of this. In such cases, you may think you're getting the benefit of a feature when in fact you aren't.

Borland's Turbo C++ is the finest implementation of a C++ compiler I have seen. This article examines how successful Borland was in implementing C++, using as example C++ 2.0 features such as multiple inheritance and pointers to members to create a "collection" class that counts keywords and identifiers in a C++ program.

Bruce is the author of Using C++ *(Osborne/McGraw-Hill, 1989), a voting member of the ANSI C++ committee (X3J16), and the owner of Revolution2, a firm specializing in C++ training and consulting. He is the C++ editor/columnist for* The C Gazette *and was a contributing writer for* MicroCornucopia *magazine for four years. Portions of this article are based on the as-yet-unpublished* The Tao of Objects, *by Bruce Eckel and Gary Entsminger.*

Surveying the Turbo C++ Landscape

Turbo C++ provides full ANSI C compatibility of Turbo C 2.0 and is fully compliant with the *C++ 2.0 Draft Reference Manual* (see the accompanying text box for a description of C++ 2.0, 2.1, and the ANSI C++ committee). Of particular interest is Turbo C++'s support for multiple inheritance, type-safe linkage, pointers to members, and built-in support for abstract classes. Borland has also implemented the AT&T C++ 2.0 iostream package, as well as the older C++ 1.0 streams (for backward compatibility with old code) and the *complex* class.

Turbo C++ comes with a library of classes intended to help you build programs, including useful classes such as *Dictionary* and *HashTable*. Class libraries address the common complaint among C++ programmers (to wit: "Fine, C++ makes code reuse easier. Where's the code to reuse?"). However, Borland made the assumption that what was good design for Smalltalk would be good design for C++. Thus we have the ponderous Smalltalk-like hierarchy, where everything is derived from a base class *Object* (no use is made of multiple inheritance), the sometimes irritating Smalltalk names such as *theQueue* and *hashValueType* (they've also made the assumption that you should be able to ask any object what type it is, rather than letting the object worry about it's own behavior), and some rather strange-looking syntax (definitely not for novices). I would have liked it better had Borland followed AT&T's lead and included a task library instead.

On the positive side, there's source code for everything, the manual is on disk, and, of course, you aren't forced to use the libraries. Simple examples show how to use some of the classes (a sorted directory listing program, for instance).

Other Features

You can easily generate assembly output with Turbo C++. This can be very helpful, especially if you want to hand-optimize code or create your own assembly language routine to link into a C++ program. The name-mangling scheme used is very easy to read (an aside: When you get linker error messages, the names are automatically unmangled for you. Nice!).

The Turbo C library is available in Turbo C++. You can link with assembly language and other libraries, including all those created for the Borland environment (that is, Turbo Pascal and Turbo Assembler) or with Microsoft C. Paradox Engine code can be linked in with Turbo C++, providing a framework for some powerful C++ database applications. And I understand you can even link to Turbo Prolog object modules (even though Turbo Prolog has reverted to the Prolog Development Center, its original creator). As usual, Borland encourages third-party support.

The *asm( )* directive in C++ is supported in Turbo C++; this allows inline assembly code. You have full control of the expansion of inline C++ functions; you can turn inlining off if you want (very convenient if you go crazy with inlines and later come to your

(continued from page 94)
senses). The compiler merges literal strings to generate smaller programs. You have full control of the way virtual tables are generated. Borland has been careful not to do anything to prohibit embedding Turbo C++ code.

Turbo C++ uses the VROOMM overlay manager for building big programs using EMS/disk overlays. Sadly, it

C++ Release 2.0

In May 1989, Bjarne Stroustrup (the creator of C++) and the team at AT&T announced C++ release 2.0 and provided the *C++ 2.0 Draft Reference Manual* (commonly referred to as the *DRM*). This was a significant upgrade to the earlier release 1.2. The largest changes were the addition of multiple inheritance and type-safe linkage (which, although invisible to the programmer, prevents one of C's more subtle and obnoxious errors), but a number of other smaller gems added to the language make a significant difference in what we can do.

Class-by-class Overloading of Operator *new* and *delete* In release 1.2, you could only change the activity of the dynamic object creation operator *new( )* (and its complement, operator *delete( )*) on a global basis. For example, overloading *new* and *delete* allows you to provide a more efficient memory-allocation scheme. This is generally something you only want to do for an individual class (one you allocate and deallocate a lot of objects for) so this is a significant improvement (also, you can forget about "assignment to this"). You can also create a garbage collector for an individual class; calling *new* for that class can register the object with your garbage collector. You can now specify the physical location in memory where you want an object to be placed (especially useful in embedded systems or other hardware-specific situations).

Support for Abstract Classes An abstract class is a base class from which you inherit a number of derived classes. It establishes the "common interface" to a set of polymorphic classes. For instance, you might create a class *sortable*, which defines a type of object that can be sorted. You never make any *sortable* objects; instead you make objects of classes derived from *sortable* (i.e., *record*). To prevent the user of a class from creating a *sortable* object, C++ 2.0 allows you to create "pure" *virtual* functions by setting the function body to zero, as in *virtual int sort( ) = 0;*. The compiler won't allow you to create any instances of a class containing a pure *virtual* function.

Copying and Initialization If you forget, or you don't want to define an *operator( )* or a copy-constructor *X(X&)* for your class, the 2.0 compiler now does it for you. These functions have always been a source of confusion for new C++ programmers, but they are important when copying an object or passing it by value into or out of a function.

More Operator Overloading You can now overload the *comma* operator and the -> operator (sometimes called a "smart pointer").

Const, Volatile, and Static Member Functions These modifiers allow you to change the way the compiler treats class member functions. A *const* member function cannot change the member data of an object, nor call another function that does. Thus it can be called for a *const* object. Similarly, a *volatile* member function can be called for a *volatile* object; the compiler must assume the object may be changed by outside forces and thus cannot make any assumptions about object stability during optimization of a *volatile* member function. A *static* member function is like an ordinary function except it is part of the class, so its name is hidden and it can only be called for that class. However, it cannot access any non-static members of an object (either functions or data).

Sophisticated Initialization Among the forms of initialization in 2.0, you can initialize automatic and global ("static") objects using all kinds of complicated expressions. For example, if *record* and *book* are both derived from *sortable*, you can write:

```
sortable * s[] = {
    new record;
    new book;
};
```

Pointers to Members You can take the offset of a class member and pass it into a function, which can then use that offset to call a member function or modify member data. This is often used in a function which acts like "apply" in Lisp — it will take the

doesn't support the equivalent of Zortech's *__handle* pointers (see "Getting a Handle on Virtual Memory," by Walter Bright, *DDJ*, May 1990) to do the same thing for data. I think the *__handle* pointer is one of Walter Bright's more terrific ideas. I'm hoping Borland will add this in the future.

The ability to set the library, *include* information, and command-line flags

member function of your choice and apply it to each object in a list. This feature has actually been part of the language before release 2.0, but has not been universally supported (Zortech 2.06, for instance, doesn't support pointers to members; Zortech 2.1 does).

C++ 2.1 and ANSI C++

Release 2.0 was to have been the specification for the language until the time that the ANSI C++ committee (X3J16) came out with ANSI C++. However, Stroustrup found some small problems with the language and changed them while in the process of writing the *Annotated C++ Reference Manual* (by Bjarne Stroustrup and Margaret Ellis, Addison-Wesley, 1990). The *ARM*, as the *Annotated Reference Manual* is called, serves as the primary base document for the ANSI C++ committee. AT&T has released version 2.1 of its cfront C-code generator (which takes C++ and generates C; these have sometimes been called translators, as opposed to native-code compilers like Turbo C++ and Zortech C++). All this happened rather suddenly, so the version of Turbo C++ I had conformed to the *DRM* (2.0) rather than the *ARM* (2.1). However, the changes are small so I expect conformance to 2.1 fairly soon; not having the changes in release 2.1 isn't inconvenient and you may not even notice them unless you already know the language quite well.

Additional Changes In 2.1

Classes, enumerations, and typedefs defined within a class are now local to the class. The conversion from a pointer to derived-class object to pointer to base-class object has been clarified. A class with a copy-constructor may be passed by value to a function with an ellipses argument (although a bitcopy is done, rather than a call to the copy-constructor). You now delete arrays with *delete [lp];*. Notice that the number of objects in the array is no longer necessary. An object which is created under a condition must be destroyed under that condition, and cannot be accessed outside that condition (for example, in *if(x) for(int i = 10; i; i--) foo(i);* you cannot access *i* for the rest of the scope).

Pure *virtual* functions are implicitly inherited as pure. Calling a pure *virtual* function inside a constructor produces undefined behavior, rather than an error message. Constructors with all default values are now considered "default" constructors (they can be used in places where a constructor with no arguments is required, for example, arrays definitions). You can define built-in types as if they had constructors; for example, *int i(5);* (similarly for destructor calls).

Resolution of function overloading has been clarified and improved. You can distinguish between pre- and postfix operator ++ and --. Unions can have *private* and *protected* members. *delete* cannot be used on a pointer to a *const*. *friend* functions are forced to be *extern* (publicly visible). Base classes may be inherited as protected. An explicit destructor call *p->X::* ~X( ) calls X's destructor even if it's virtual, while *p->* ~X( ) uses the *virtual* mechanism.

Your Comments and Suggestions

The ANSI C++ committee had its first meeting March 12 – 16 in New Jersey and the second July 9 – 12 in Seattle. Written public suggestion and comment is invited (in fact, it's an integral part of the process). The committee expects to finish its work in roughly three years (since it has a running start with the *ARM* and the ANSI/ISO C specs). The major extensions that will likely be added by the ANSI committee include exception handling and parameterized types (implementations of which you may see before the committee is complete). Send suggestions and comments to:

Dmitry Lenkov
X3J16 Committee Chair
HP California Language Lab
19447 Pruneridge Avenue, MS: 47LE
Cupertino, CA 95014
email:dmitry%hpda@hplabs.hp.com

— B.E.

in the TURBOC.CFG file is excellent! This immediately eliminates all the path collision and environment-space problems you have when running more than one C++ compiler on your machine (I do this when I want to test correctness). Turbo C++ will also swap the symbol table to extended memory or EMS for compilations.

Although (as with all C++ implementations) there are limitations on what the C++ compiler will actually inline, there are far fewer restrictions on what the compiler will accept in an inline statement. For instance, Turbo C++ would accept static variables and "embedded" return statements (those that aren't at the end of the function) in inline functions, while Zortech and cfront wouldn't.

You can't always determine how many and what type of objects you need while you're writing a program

Programmer's Platform

A Borland language product wouldn't be complete without the Integrated Development Environment (IDE). The IDE includes everything you need to edit, compile, link, and debug programs. This time, however, the IDE has been completely reworked with multiple overlapping windows you can reshape and move around the screen. Other features include a hypertext-style help system, the ability to copy code fragments from the help system to the editor, and more. For those of you who can't seem to get by without rodents, there's full mouse support. And, a nice interactive tour teaches you how to use the IDE features.

The "Turbo Editor Macro Language" (TEML), a very readable scripting language that looks like a bastardization of Pascal and C, lets you use macro scripts to write customizations. For instance, you can rebind all your keys when the editor starts up, grab class names and member function prototypes, pop to another file and append the class name to a :: and the function prototype to create a definition stub, and so on. The macros are compiled with the "Turbo Editor Macro Compiler" (TEMC), which avoids many of the speed problems associated with interpreted macro languages. You can create named macros (with arguments) and you can bind those macros to keys; you can also just tie a set of statements to a key without a name. There are over 140 built-in commands to work with. Example 1 shows a simple example of a macro which puts void at the beginning of a line, then returns to the point where you invoked it. Unfortunately, it is difficult to determine from the documentation the limitations of TEML.

The IDE also has a built-in, small version of the Turbo Debugger, which displays C++ source code as you've typed it in (no messes from name mangling) and allows you to perform various debugging feats.

Turbo Professional

The full-blown debugger (provided with the Turbo Professional, or sold separately with the Turbo Debugger and Tools package) is a step forward in debugger technology. It's better than anything I've seen (although I admit I never had the patience to figure out CodeView). I didn't need to use the manual much at all, since the menus and help system are so good. The debugger can use 80386 protected mode by loading a special device driver; this allows viewing assembly language after a hard crash. I particularly like the traceback feature; you can use the animation feature to automatically step until the program crashes, then trace back and see its last thoughts (although it would be nice if you could speed

```
macro InsertVoid
      SetMark(0);  /* save our place */
      LeftOfLine;
      InsertText("void ");
      MoveToMark(0);  /* restore the place */
end; /* end of macro InsertVoid */

Alt-V : InsertVoid;   /* bind to key */
```

Example 1: *Macro that puts void at the beginning of a line, then returns to the point where you invoked it.*

(continued from page 98)
up animation in this process by eliminating screen updates). The display of class hierarchy is fine, but it isn't much use by itself; perhaps the debugger is the wrong place for it — you want to see hierarchies while you're writing code, not while you're debugging it.

The Turbo Profiler (also part of the professional package) should alleviate any concerns you have about potential performance differences with C. It seems to me the benefit of optimizers is overstated. An optimizer doesn't have enough information to make a large difference in execution speed. If you can find the place where your program is spending all its time, you can go in and optimize that section by hand, rather than letting the compiler sprinkle potentially useless speed improvements throughout the program. You can think of the profiler as attacking the problem from the opposite end, and the improvements in speed are potentially far more dramatic than what you will see with an optimizer. Don't underestimate the value of this tool.

Taking Up A Collection

The code presented in this section demonstrates several concepts programmers new to C++ should understand. These concepts include:

- Collections, which support a dynamic style of programming;
- Multiple inheritance, so that classes can be derived from more than one base;
- Pointers to members, which let you delay the selection of a member (function or data) until run time.

The classes created here are simple but powerful. The example that illustrates them is used to count the keywords and identifiers in a C++ source file.

A collection is an object which holds an arbitrary number of other objects. Basically, it's just a bag you can throw things into and fish them out later. The reason collections are so important is that they support a dynamic style of programming. You can't always determine how many and what type of objects you need while you're writing a program (although you may be in the habit of trying to figure it out). In the general case, these things can only be established at run time. Thus, objects must be created and destroyed on-the-fly, using dynamic object creation (via the operators *new* and *delete*). While you're working with these objects, you need some place to stash them — that's where the collection comes in.

You're probably familiar with linked lists, stacks, queues, and trees; these are all prototypical collections in non-object-oriented languages. The collection in Listing One, page 132, looks vaguely like a singly-linked list. I like it precisely because it is so simple, but you may have to stare at it a while to figure out exactly how it works.

The first definition is a *struct* called *item*, which only contains a virtual destructor. Any object you want to put into a collection should be derived from *item* so that *collection* can "own" the items it holds: When it wants to destroy the item, the proper destructor is called (note that ownership isn't always so clear cut — in some situations an object is contained in more than one list).

A really good class library is carefully designed and redesigned and tested through use

A *collection* is a chain of *holder* objects. Each *holder* is a link in the chain; it has a pointer to an *item* and a pointer to the next *holder*. Initializing a *holder* is simply setting its two pointers. Note that although *struct holder* is defined inside *class collection*, *holder* is a global name in C++ 2.0 (but hidden in C++ 2.1).

A *collection* object just contains two *holder* pointers — one called *head*, which points to the top of the list, and one called *cursor*, which moves through the list and points to the *holder* we're currently interested in. Initializing a *collection* means setting its two pointers to *NULL* (to indicate the list is empty). Cleaning up the list is more complex; we must go to the *head* and move through the list until it's empty, deleting each *item* and *holder* in the list. Here you'll need to ponder the code and the comments, imagining what's going on. Sometimes it helps to draw pictures.

```
item * i = c.reset();
while(i) {
  // do something
  i = c.next();
}
```

Example 2: *Stepping through a typical collection.*

Adding a new *item* to a *collection* is the slickest part of the whole class. Make a new *holder* using the current *head* pointer and the new *item* pointer as arguments. The value returned by *new* (the address of the newly created *holder*) becomes the new value for *head*. In one statement the list expands, just like yeast budding a new cell (this is the *add( )* function). Because *add( )* is inline, adding an element is amazingly fast.

Stepping Through a Collection

You step through a collection using the *reset( )* and *next( )* commands, each of which returns a pointer to the current *item*, or *NULL* if we're at the end of the list. For a *collection c*, use the commands as shown in Example 2. If the list is empty, *reset( )* returns *NULL* and the while loop is never entered. Note that in both *reset( )* and *next( )*, we must take care not to return an *item** if cursor is *NULL*; this is taken care of by the ternary if-else statement (? and :). In *next( )*, we must not move forward if cursor is *NULL*.

Multiple Inheritance

In a "pure" object-oriented language such as Smalltalk, everything is an object. All objects have as their base the same class (called "object" or "root" or something like that). Collections in Smalltalk are usually part of the existing system; you don't have to make them yourself. They are designed to hold any type of object, thus they usually hold root objects, or something close to root in the inheritance tree.

The need for multiple inheritance is not obvious in a system like this; multiple inheritance allows you to combine the characteristics of two or more classes, but in Smalltalk all objects already have a lot of things in common. Now, consider C++ where classes are often made from scratch instead of being inherited from some master root class. What happens if you have two predefined classes (which you can't or don't want to change) that must be combined into one? As an example, consider a relatively simple class that holds a word and compares it to text strings, incrementing a counter if there is a match. Since we're defining the *word* class right here we could have inherited it from *item*, but imagine it has been created by someone else and is much more complicated.

Now suppose you want to make a collection of *word* objects. Anything that goes in a collection must be derived from *item* so the collection can "own" it and properly destroy it. But *word* has already been created, so it

can't be derived from *item*. What we need is a way to derive a new type from *word* and *item* at the same time. That's where multiple inheritance is essential in C++. You can see it used in the creation of class *worditem*, which is a word that can also be stored in a collection. The first *main( )* in Listing One is a simple demonstration of a collection of *worditems*.

Counting Words

If we want to count instances of words instead of just adding each word to the list, we need to modify the *collection* class by inheriting it into a new class called *wordcounter*. This class is first customized so that *add( )* only accepts *worditem* pointers, and *reset( )* and *next( )* only return *worditem* pointers. Although this may look as if you're adding code, there's actually no overhead — the compiler does all the work by enforcing type checking. We then add a new function, *add_or_count( )*, to class *wordcounter*. This will test a string against each word in the list; if it finds a match, that word gets incremented; if not, it adds the word to the list.

Pointers to Members

Finally, we add a function called *apply*, which uses pointers to members. These are particularly useful for collections — in *apply( )*, the function argument is applied to each member of the list. Note that a pointer to a member function looks a lot like a pointer to a function; it just has the class name and the scope resolution operator added. The *apply( )* function is demonstrated in the second *main( )*.

You can imagine a more powerful construct: Consider an array of pointers to member functions. You could index into this array to select the desired member. Pointers to members can also be used to change the behavior of a "call back." For example, with a mouse event that might cause the system to change state, the next mouse event should cause something different to happen.

Counting Identifiers and Keywords

Now that we have the desired class and the desired collection, let's use them to count the identifiers and keywords in a C++ source code file (I tested it on Listing One). The trick is simply to remove all the quoted strings, operators, comments, and constants and put the remaining words into our *wordcounter* collection by using *add_or_count( )*. This is accomplished by using the ANSI C library functions *strchr( )*, which finds the first instance of a character in a string, *strstr( )*, which finds the first instance of a string within a string, *strtok( )*, which breaks a string up into tokens according to the delimiters of your choice, and *strcspn( )*, which counts the characters in the initial length of a string that doesn't consist of characters from your chosen string; this is used to determine if a string is a numerical constant. Look these up in your ANSI C library reference for further details.

I'm using standard I/O here instead of bothering to open and close files. You have to redirect input and output on the command line, but it makes the example simpler.

In running TEST3, notice that the results of the analysis are printed to standard output as the destructors are called for the *worditem* objects (the print statement is in the *word* destructor).

You can easily create a second type of list to hold keywords. Initialize this list with the C++ keywords, then test each new word against this list before conditionally adding it to the *symbol_table* list. This way you can separate keywords and identifiers.

Note that parts of Listing One will compile with Zortech C++ 2.06 (if you remove the pointers to members, the whole thing will compile). However, the TEST1 code shows a place where Zortech C++ 2.06 has a bug (it works properly with cfront 2.0 and Turbo C++).

I believe Borland will capture the C++ market in much the same way they did the Pascal market. They are just out of the blocks with Turbo C++ and already far ahead of everyone else. Other C++ vendors will have to come up with something pretty spectacular for me to pry my clenched fingers from Turbo C++.

DDJ

Products Mentioned:

Turbo C++
Borland International
P.O. Box 660001
Scotts Valley, CA 95066-0001
$199.95
$299.95 for Turbo C++ Professional
Special prices for registered owners of Turbo C.

(Listing begins on page 132.)

Vote for your favorite feature/article.
Circle Reader Service **No. 9.**

Listing One *(Text begins on page 16.)*

```
; William F. Dudley, Jr.
; real mode module callable from protected mode, passes string in
; buffer in code segment.
ss_text         SEGMENT BYTE PUBLIC 'CODE' use16
                ASSUME  CS:ss_text , DS:NOTHING , ES:NOTHING
; The string is pointed to by DS:SI with the length in CX.
; encryption value of string returned in AX.
                public  QUERY_FAR

QUERY_FAR       PROC    FAR
                CALL    QUERY
                RET
QUERY_FAR       ENDP

QUERY           PROC    NEAR
;
; here lives the real mode dongle communications code
QUERY           ENDP

                public  _test_string
_test_string    db      256 dup (?)

                public  _end_real
_end_real label byte

ss_text         ENDS
                END
```

End Listing One

Listing Two

```
/* William F. Dudley Jr.
 * module to connect real mode assembly code to prot mode C
 */
#include <stdio.h>
#include <dos.h>
#include "list5.h"

int real_addr;                  /* real address of start of real mode code */
int real_seg, real_off;         /* segment and offset of real_addr */

#pragma aux QUERY_FAR "*";      /* tell Watcom C that no '_' is used */

extern char test_string;        /* string buffer in real mode module */
extern char end_real;           /* end of real mode code */
extern char QUERY_FAR;  /* actually a function, but we just want address */

short COMPUTE(char *str);       /* this is the subroutine that does the work */
int pr2real(void);              /* initialize value of real_addr, etc. */

void real_setup(void) {
    real_addr = pr2real();
    real_seg = (real_addr >> 16) & 0xffff ;
    real_off = real_addr & 0xffff ;
}

int pr2real(void) {
union REGS r;
struct SREGS sr;

  r.x.ax = 0x250f;      /* convert prot addr to real addr */
  r.d.ebx = 0;          /* start of program */
  r.d.ecx = (int)&end_real;     /* length of real mode stuff */
  sr.fs = 0x3c;
  sr.es = Ds();
  sr.ds = sr.ss = sr.gs = 0x14;
  sr.cs = 0x0c;
  int386x(0x21, &r, &r, &sr);
  if(r.d.cflag) {
      fprintf(stderr, "Error in PR2REAL(), can't map address.\n");
      fflush(stderr);
      exit(1);
  }
  return(r.d.ecx);
}

short COMPUTE(char *str) {
union REGS r;
struct phregs pr;
unsigned short i;
unsigned j;
  r.x.ax = 0x2510;              /* call function */
  r.d.ebx = real_addr;          /* get segment of real mode stuff */
  r.x.bx = (int)&QUERY_FAR;     /* EBX is address of QUERY_FAR subroutine */
  r.d.ecx = 0;                  /* 0 words on stack */
  r.d.edx = (int)&pr;           /* DS:EDX is address of register struct */
  pr.ECX = strlen(str);         /* CX is length of string */
  i = real_off + (int)&test_string;
  j = i + (real_seg<<4);              /* calculate address in selector 0x34 */
  blk_mv_pr(str, 0x34, j, pr.ECX);    /* copy string to buffer in real CS */
  r.x.si = i;                   /* DS:SI points to string */
  pr.ES = pr.DS = real_seg;
  int386(0x21, &r, &r);
  return(r.x.ax);
}
```

End Listing Two

Listing Three

```
; William F. Dudley Jr.
; copy from real to protected or vice-versa
; void blk_mov_pr(char *bufadr, unsigned reg_seg, unsigned reg_off, unsigned
count);
; type  variable        is in:
; char  *bufadr         EAX
; uint  reg_off         EBX
; uint  count           ECX
; uint  reg_seg         EDX
; Transfers COUNT bytes from the buffer (in the current data seg) bufadr
; to address in protected memory at reg_seg:reg_off.

        NAME    blk_mov
        EXTRN   __STK:WORD
_TEXT   SEGMENT PUBLIC BYTE USE32 'CODE'
        ASSUME  CS:_TEXT
        PUBLIC  blk_mov_pr_
        PUBLIC  blk_mov_rp_

; protected to real
blk_mov_pr_     proc    near
        pushf
        push    EDI
        push    ESI
        push    ES
        jecxz   non1
        cld
        ; count is in ECX already
        mov     ESI, EAX        ;bufadr is source
        mov     ES, DX          ;reg_seg is dest (ES:EDI)
        mov     EDI, EBX        ;reg_off is dest
        rep     movsb
non1:   pop     ES
        pop     ESI
        pop     EDI
        popf
        ret
blk_mov_pr_     endp

; real to protected
blk_mov_rp_     proc    near
        pushf
        push    EDI
        push    ESI
        push    ES
        push    DS
        jecxz   non2
        cld
        push    DS
        pop     ES
        ;count is in ECX
        mov     EDI,EAX         ;bufadr is dest (ES:EDI)
        mov     DS,DX           ;reg_seg is source (DS:ESI)
        mov     ESI,EBX         ;reg_off is source
        repe    movsb
non2:   pop     DS
        pop     ES
        pop     ESI
        pop     EDI
        popf
        ret
blk_mov_rp_     endp

_TEXT   ENDS
        END
```

End Listing Three

Listing Four

```
/* William F. Dudley Jr. */

#include <stdio.h>
#include <dos.h>
#include <process.h>
#include <io.h>

#include "list5.h"              /* dud's driver interface constants */

/* map of real mode link (call buffer) memory:
 * rel address  name            comment
 * 0            rcolortable     pointer to 128 bytes for color table storage
 * 128          qpixel          pointer to MAXB (8500) bytes for pixel storage
 */
#define MAXB 8500

int mblocks;                    /* no of save/restore blocks    */
static int ddi_allocated = 0;   /* true if initialization has been performed */
static int rcolortable; /* real mode address of colortable (intermode buf) */

static int qpixel;              /* real mode address of line storage space */
int nsegment, noffset;          /* line storage segment & offset */
static short psegment;          /* protected address of pixel buffer */
static int poffset;             /* protected address of pixel buffer */

extern int vectnum;             /* ddi interrupt vector (usually 0x7A) */
extern void pdintinit(void);    /* setup for pdinterp() */

static int blocksizes[100];     /* array of saved blocks for save/rstr scrn */

/* sets video mode and initializes the video parameters */
void setvmode(void)
{
  union REGS r;
  struct SREGS sr;
  int i;
```

(Listing continued on page 106)

Listing Four *(Listing continued, text begins on page 16.)*

```
char paltbl[64];      /* driver will dump palette here. */

   if(!ddi_allocated) {
       r.x.ax = 0x250d;        /* get real mode link information */
       /* segment regs must all have legal values in them.
        * These values are documented in the Phar-Lap Extender manual
        */
       sr.fs = 0x3c;
       sr.ds = sr.ss = sr.es = sr.gs = 0x14;   /* the data "segment" */
       sr.cs = 0x0c;                           /* the code "segment" */
       int386x(0x21, &r, &r, &sr);
       /* es:edx = protected address of call buffer */
       rcolortable = r.d.ebx;  /* ebx = real address of call buffer */
       poffset = r.d.edx;      /* save protected offset to table start */
       psegment = sr.es;       /* psegment = 0x60 in Phar-World */
       qpixel = rcolortable + 128;
       reset_lines();
       if(r.d.ecx < (MAXB + MAXCOLORS)) {      /* ecx = size of buffer */
           fprintf(stderr,"real mode buffer isn't big enough: %d\n", r.d.ecx);
           abort();
       }
   }
   r.h.ah = KINIT1;
   r.x.bx = (rcolortable >> 16) & 0xffff;      /* segment of real mode buf */
   r.x.cx = rcolortable & 0xffff;              /* offset of real mode buf */
   r.x.si = r.x.di = 0;        /* clear so we can tell if they are set */
   int86(vectnum, &r, &r);
   /* The registers have various video constants in them.  The code that
    * uses them is not shown for clarity.
    */
   if(!r.x.si && !r.x.di) {    /* if driver does not return its address */
       fprintf(stderr, "old driver installed, you need current version!\n");
       exit(1);
   }
   prot.vidfn.addr[1] = r.x.si;        /* real mode address of video entry */
   prot.vidfn.addr[0] = r.x.di;
   pdintinit();
   listinit();
   color_mask = (int)r.h.al - 1;
   if(!ddi_allocated) {
       /* copy from real to prot bufr */
       blk_mv_rp(paltbl, psegment, poffset, 64);
       /* copy array of chars to array of ints */
       for(i=0 ; i <= color_mask ; i++ ) colortable[i] = (int)paltbl[i];
   }

   r.h.ah = KINIT2;
   r.h.al = 0;                 /* don't switch modes */
   int86 (vectnum, &r, &r);
   /* The registers have various video constants in them.  The code that
    * uses them is not shown for clarity.
    */
   mblocks = r.h.dl;           /* number of blocks to save screen */
   ddi_allocated = TRUE;
}

void reset_lines ()
{
   nsegment = (qpixel >> 16) & 0xffff;
   noffset = qpixel & 0xffff;
   return;
}

/* Restore Video Buffer, returns status */
int rstr_vbuf(void)
{
union REGS r;
int i, l;
int rtncode = 0;
char bbuf[8200];
char *lbuf;
lbuf = (char *)bbuf;

   if (vbfnum!=NULL) rtncode=lseek(vbfnum,0L,0);       /* beg of file */
   else return(OKAY);

   r.h.ah = INITDMP;           /* init driver */
   r.h.al = SWRITE;
#ifdef __386__
   r.x.bx = nsegment;
   r.x.cx = noffset;
#else
   r.x.bx = FP_SEG(lbuf);
   r.x.cx = FP_OFF(lbuf);
#endif
   int86(vectnum, &r, &r);
   /* now restore screen */
   for(i = 0 ; i < mblocks ; i++ ) {
       rtncode=read(vbfnum, lbuf, blocksizes[i]);
       if(rtncode<= 0) return(ERROR);
       r.h.ah = KDUMP;
#ifdef __386__
       l = (blocksizes[i] < 8192) ? blocksizes[i] : 8192 ;
       blk_mv_pr(bbuf, psegment, poffset+128, l); /* copy from prot to real */
#endif
       int86(vectnum, &r, &r);
   }
   return(OKAY);
}

/* clear the draw list */
void listinit(void) {
   prot.list[0][0] = 0;
   prot.lp = 0;
   list_p = prot.list[0];
}
```

End Listing Four

(Listings continued on page 108.)

Listing Five *(Listing continued, text begins on page 16.)*

```
/* William F. Dudley, Jr.
 * macros for getting to the driver from an application
 */

extern int vectnum;

int Ds(void);                   /* what is value of DS register */
#pragma aux Ds = \
        0x8c 0xd8       /* mov ax, ds */ \
        modify [AH AL];

/* register arrangement for Phar-Lap function 0x2510 */
struct phregs {
        unsigned short DS;
        unsigned short ES;
        unsigned short FS;
        unsigned short GS;
        int EAX;
        int EBX;
        int ECX;
        int EDX;
        } ;

#define LLEN 100        /* assembly language module must agree with this */
#ifndef PDINTERP
extern
#endif
        struct {
        union {
            int (* p)(); /* this is for human info only, we never call it */
            short int addr[2];  /* [0]seg and [1]off of driver entry point */
        } vidfn ;
        short int lp;
        short int list[LLEN][6];
    } prot ;
#ifndef PDINTERP
extern
#endif
        short int *list_p;
void listinit(void);
void pdinterp(void);

void kdidraw(short int,short int,short int,short int,short int,short int);
/* tell Watcom C how to use registers for arguments */
#pragma aux kdidraw parm [EAX] [EBX] [ECX] [EDX] [ESI] [EDI];

/* move to x1,y1, route/line width will be w */
#define M(x1,y1,w) kdidraw((KMOVE<<8), x1, y1, w, 0, 0)
/* put dot at x1, y, color c, atrib at */
#define DOT(x1,y1,c,at) kdidraw(at+(KWDOT<<8), x1, y1, c, 0, 0)
/* draw line from M point to x1,y1, color c, atrib at */
#define D(x1,y1,c,at) kdidraw(at+(KDRAW<<8), x1, y1, c, 0, 0)
```

End Listing Five

Listing Six

```
; William F. Dudley, Jr.
; "porting a large application to 386 protected mode"
; This is the protected mode function that stuffs draw commands
; in the draw list.  If the list fills up, it automatically calls
; pdinterp() to empty it.
;
            NAME    storlist
            EXTRN   pdinterp_:WORD
            EXTRN   _prot:WORD
            EXTRN   _list_p:WORD
LLEN    EQU     100             ; size of draw list, must agree with C version.
DGROUP      GROUP   CONST,_DATA,_BSS
_TEXT       SEGMENT PUBLIC BYTE USE32 'CODE'
            ASSUME  CS:_TEXT,DS:DGROUP
            PUBLIC  kdidraw_
            ; args in ax,bx,cx,dx,si,di
            ; global list pointer in _list_p is incremented by 12
kdidraw_:
            push    esi                         ;save si
            mov     si,ax                       ;save ax
            mov     eax,dword ptr _list_p
            mov     word ptr [eax],si
            mov     word ptr [eax+2],bx
            mov     word ptr [eax+4],cx
            mov     word ptr [eax+6],dx
            pop     esi                         ; get back si
            mov     word ptr [eax+8],si
            mov     word ptr [eax+10],di
            add     eax, 12
            mov     dword ptr _list_p,eax

            inc     word ptr _prot+4H
            cmp     word ptr _prot+4H,LLEN-3
            jle     L1
            call    near ptr pdinterp_
            jmp     short L2
L1:         mov     word ptr [eax],0000H
L2:         ret
_TEXT       ENDS

CONST       SEGMENT PUBLIC WORD USE32 'DATA'
CONST       ENDS
_DATA       SEGMENT PUBLIC WORD USE32 'DATA'
_DATA       ENDS
_BSS        SEGMENT PUBLIC WORD USE32 'BSS'
_BSS        ENDS
            END
```

End Listing Six

Listing Seven

```
/* pdinterp.c -- William F. Dudley, Jr.
 * protected dinterp() for Phar-Lap environment.
 * this is the protected half of the draw list kdi processor
 */
#include <stdio.h>
#include <dos.h>
#define PDINTERP 1
#include "list5.h"

extern int real_addr;           /* real address of start of real mode code */
extern int real_seg, real_off;          /* segment and offset of real_addr */
extern char real;               /* real copy of vidfnp, lp, draw list */
extern char end_real;
extern char dinterp;    /* actually a function, but we just want address */
void pdinterp(void);
int pr2real(void);

static union REGS pregs;
static struct phregs pr;
static unsigned short real_o;
static unsigned abs_adr;
static int pdinitted=0;

void pdinterp() {
union REGS r;
  if(!prot.lp) return;
  if(!pdinitted) pdintinit();
  /* copy list to buffer in real code seg */
  blk_mov_pr(&prot, 0x34, abs_adr, sizeof(prot));
  int386(0x21, &pregs, &r);
  prot.list[prot.lp = 0][0] = 0;
  list_p = prot.list[0];
}

void pdintinit(void) {
  pregs.x.ax = 0x2510;          /* call function */
  pregs.d.ebx = real_addr;      /* get segment of real mode stuff */
  pregs.x.bx = (short int)&dinterp; /* EBX is address of dinterp subroutine */
  pregs.d.ecx = 0;              /* 0 words on stack */
  pregs.d.edx = (int)&pr;       /* DS:EDX is address of register struct */
  real_o = real_off + (short int)&real;
  abs_adr = real_o + (real_seg<<4);   /* calculate address in selector 0x34 */
  pregs.x.si = real_o+6;              /* DS:SI points to list */
  pr.ES = pr.DS = real_seg;
  pdinitted = 1;
}

#if IN_C
/* this is a C version of the kdidraw() function in list6.asm */
void kdidraw(short int Ax,short int Bx,short int Cx,short int Dx,
                                                  short int Si,short int Di) {
register short int *pi_;
    pi_=prot.list[prot.lp];
    *pi_++ = (short)(Ax);
    *pi_++ = (short)(Bx);
    *pi_++ = (short)(Cx);
    *pi_++ = (short)(Dx);
    *pi_++ = (short)(Si);
    *pi_++ = (short)(Di);
    if(++prot.lp > LLEN-3) pdinterp();
    else *pi_ = 0;
}
#endif
```

End Listing Seven

Listing Eight

```
; William F. Dudley, Jr.
; "Porting a large application to 386 protected mode"
; This is the real mode draw list interpreter.
;
d_text  SEGMENT BYTE PUBLIC 'CODE' use16
        ASSUME  CS:d_text , DS:NOTHING , ES:NOTHING

LLEN    EQU     100             ; size of draw list

        public  _dinterp
        public  _real
_real   label   word
        db      (12*LLEN+6) dup (?)

_dinterp proc   far
        call    dint
        ret
_dinterp endp

; ds:si points to real array at entry
dint    proc    near
floop:  push    word ptr [si+10]        ; di
        push    word ptr [si+8]         ; si
        push    word ptr [si+6]         ; dx
        push    word ptr [si+4]         ; cx
        push    word ptr [si+2]         ; bx
        push    word ptr [si+0]         ; ax
        add     si,12
        call    dword ptr cs:[_real]
        add     sp,12
iftest: cmp     word ptr [si+0] ,0
        jne     floop
        ret
dint    endp

d_text  ends
        end
```

End Listings

Listing One *(Text begins on page 24.)*

```
/* junk.c  -- Junk list build/destroy test
 *  $Log:   E:/vcs/junk/junk.c_v  $
 *           Rev 1.1   20 Nov 1989 09:42:00   set
 *        Added name field for junk node tagging
 *           Rev 1.0   09 Nov 1989 18:12:30   jvs
 *        Initial revision.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Junk item structure  */
typedef struct jnod {
  struct jnod     *junk_next ;
  char            *junk_name ;
  int             junk_data[3000] ;
  } JUNK ;

/* Function prototypes  */
JUNK      *junk_list_build(void) ;
void      junk_list_destroy(JUNK *) ;
JUNK      *junk_open(char *) ;
void      junk_close(JUNK *) ;

/* main() -- Entry point for test  */
void main()
{
JUNK *jlist ;

while ((jlist = junk_list_build()) != NULL)
  {
  junk_list_destroy(jlist) ;
  }
printf("*** Should never get here! ***\n") ;
}

/* junk_list_build()  -- Build a list of junk items  */
JUNK *junk_list_build()
{
JUNK *jlist ;
JUNK *jnew ;

jlist = NULL ;
while ((jnew = junk_open("name to identify junkitem")) != NULL)
  {
  jnew->junk_next = jlist ;
  jlist = jnew ;
  }

return jlist ;
}

/* junk_list_destroy()  -- Destroy a list of junk items  */
void junk_list_destroy(JUNK *jlist)
{
JUNK *jtmp ;

while (jlist != NULL)
  {
  jtmp = jlist ;
  jlist = jlist->junk_next ;
  junk_close(jtmp) ;
  }
}

/* junk_open() -- Create a junk item  */
JUNK *junk_open(char *name)
{
JUNK *jnew ;

jnew = (JUNK *) malloc(sizeof(JUNK)) ;
if (jnew != NULL)
  {
  jnew->junk_name = strdup(name) ;
  if (jnew->junk_name == NULL)
          {
          return NULL ;
          }
  }

return jnew ;
}

/*  junk_close() -- Close a junk item  */
void junk_close(JUNK *jold)
{
free(jold) ;
}
```

End Listing One

Listing Two

```
/* junk.c -- Junk list build/destroy test
 *  $Log:   E:/vcs/junk/junk.c_v  $
 *           Rev 1.2   28 Nov 1989 10:13:07   jvs
 *        Addition of memory shell
 *           Rev 1.1   20 Nov 1989 09:42:00   set
 *        Added name field for junk node tagging
 *           Rev 1.0   09 Nov 1989 18:12:30   jvs
 *        Initial revision.
 */

#include <stdio.h>
#include <stdlib.h>
#include "mshell.h"

/* Junk item structure  */
typedef struct jnod {
  struct jnod     *junk_next ;
  char            *junk_name ;
  int             junk_data[3000] ;
  } JUNK ;

/* Function prototypes */
JUNK      *junk_list_build(void) ;
void      junk_list_destroy(JUNK *) ;
JUNK      *junk_open(char *) ;
void      junk_close(JUNK *) ;

/* main() -- Entry point for test  */
void main()
{
JUNK *jlist ;

while ((jlist = junk_list_build()) != NULL)
  {
  junk_list_destroy(jlist) ;
  if (Mem_Used() != 0)
          {
          printf("*** Memory list not empty ***\n") ;
          Mem_Display(stdout) ;
          exit(1) ;
          }
  }
printf("*** Should never get here! ***\n") ;
}

/* junk_list_build() -- Build a list of junk items  */
JUNK *junk_list_build()
{
JUNK *jlist ;
JUNK *jnew ;

jlist = NULL ;
while ((jnew = junk_open("name to identify junkitem")) != NULL)
  {
  jnew->junk_next = jlist ;
  jlist = jnew ;
  }

return jlist ;
}

/* junk_list_destroy() -- Destroy a list of junk items  */
void junk_list_destroy(JUNK *jlist)
{
JUNK *jtmp ;

while (jlist != NULL)
  {
  jtmp = jlist ;
  jlist = jlist->junk_next ;
  junk_close(jtmp) ;
  }
}

/* junk_open() -- Create a junk item  */
JUNK *junk_open(char *name)
{
JUNK *jnew ;

jnew = (JUNK *) malloc(sizeof(JUNK)) ;
if (jnew != NULL)
  {
  jnew->junk_name = strdup(name) ;
  if (jnew->junk_name == NULL)
          {
          return NULL ;
          }
  }

return jnew ;
}

/* junk_close() -- Close a junk item  */
void junk_close(JUNK *jold)
{
free(jold) ;
}
```

End Listing Two

Listing Three

```
/*---------------------------------------------------------------------
 *++
 *  mshell.h -- Dynamic memory handler interface
 *  Description: mshell.h provides the interface definitions for the
 *  dynamic memory handler.
 *  See mshell.c for complete documentation.
 *+-
 *  $Log$
 *--
 */

/* Compilation options */
#define MEM_LIST          /* Build internal list */
#define MEM_WHERE         /* Keep track of memory block source */

/* Interface functions */
unsigned long     Mem_Used(void) ;
void              Mem_Display(FILE *) ;

/* Interface functions to access only through macros */
#if defined(MEM_WHERE)
```

```
22: void             *mem_alloc(size_t, char *, int) ;
23: void             *mem_realloc(void *, size_t, char *, int) ;
24: void             mem_free(void *, char *, int) ;
25: char             *mem_strdup(char *, char *, int) ;
26: #else
27: void             *mem_alloc(size_t) ;
28: void             *mem_realloc(void *, size_t) ;
29: void             mem_free(void *) ;
30: char             *mem_strdup(char *) ;
31: #endif
32:
33: /* Interface macros */
34: #if !defined(__MSHELL__)
35: #if defined(MEM_WHERE)
36: #define malloc(a)       mem_alloc((a),__FILE__,__LINE__)
37: #define realloc(a,b)    mem_realloc((a),(b),__FILE__,__LINE__)
38: #define free(a)         mem_free((a),__FILE__,__LINE__)
39: #define strdup(a)       mem_strdup((a),__FILE__,__LINE__)
40: #else
41: #define malloc(a)       mem_alloc(a)
42: #define realloc(a, b)   mem_realloc((a),(b))
43: #define free(a)         mem_free(a)
44: #define strdup(a)       mem_strdup(a)
45: #endif
46: #endif
47:
48: /*-------------------------------------------------------------------*/
```

End Listing Three

Listing Four

```
 1: /*-------------------------------------------------------------------
 2:  *++
 3:  *  mshell.c -- Memory management utilities
 4:  *  Description: mshell.c contains routines to protect the programmer
 5:  *  from errors in calling memory allocation/free routines. The standard
 6:  *  library calls supported are: malloc, realloc, strdup, free
 7:  *
 8:  *  The interface header mshell.h redefines these standard libarary
 9:  *  calls as macros. When the client code is compiled, the macros expand
10:  *  to calls to this module. This module then calls the system memory
11:  *  routines, with additional error checking.
12:  *
13:  *  The allocation routines in this module add a data structure to the
14:  *  top of allocated memory blocks which tag them as legal memory blocks.
15:  *
16:  *  When the free routine is called, the memory block to be freed is
17:  *  checked for legality. If the block is not legal, the memory list
18:  *  is dumped to stderr and the program is terminated.
19:  *
20:  *  Compilation Options
21:  *  MEM_LIST  Link all allocated memory blocks onto an internal list.
22:  *    The list can be displayed using Mem_Display().
23:  *  MEM_WHERE Save the file/line number of allocated blocks in header.
24:  *    Requires that compiler supports __FILE__ and __LINE__ preprocessor
25:  *    directives and __FILE__ string have static or global scope.
26:  *  Problems: If you have any problems with this module, please contact:
27:  *    Jim Schimandle, Primary Syncretics, 473 Sapena Court, Suite #6,
28:  *    Santa Clara, CA 95054, Tel: (408)988-3818, Fax: (408)727-9891
29:  *+-
30:  *  $Log$
31:  *--
32:  */
33:
34: #define __MSHELL__
35:
36: #include <stdio.h>
37: #include <stdlib.h>
38: #include <string.h>
39: #include "mshell.h"
40:
41: /* Constants */
42: #define MEMTAG    0xa55a                /* Value for mh_tag */
43:
44: /* Structures */
45: typedef struct memnod                   /* Memory block header info     */
46:   {                                     /* ---------------------------- */
47:   unsigned int    mh_tag ;              /* Special ident tag            */
48:   size_t          mh_size ;             /* Size of allocation block     */
49: #if defined(MEM_LIST)
50:   struct memnod   *mh_next ;            /* Next memory block            */
51:   struct memnod   *mh_prev ;            /* Previous memory block        */
52: #endif
53: #if defined(MEM_WHERE)
54:   char            *mh_file ;            /* File allocation was from     */
55:   unsigned int    mh_line ;             /* Line allocation was from     */
56: #endif
57:   } MEMHDR ;
58:
59: /* Alignment macros -- The macro ALIGN_SIZE defines size of the largest
60:  * object that must be aligned on your processor. The RESERVE_SIZE macro
61:  * calculates the nearest multiple of ALIGN_SIZE that is larger than the
62:  * memory block header. Change ALIGN_SIZE to match alignment requirements
63:  * of your processor
64:  */
65: #define ALIGN_SIZE sizeof(double)
66:
67: #define HDR_SIZE sizeof(MEMHDR)
68: #define RESERVE_SIZE (((HDR_SIZE+(ALIGN_SIZE-1))/ALIGN_SIZE) \
69:                       *ALIGN_SIZE)
70:
```

(Listing continued on page 112)

Listing Four *(Listing continued, text begins on page 24.)*

```
/* Conversion macros -- These macros convert the internal pointer
 * to a memory block header to/from the pointer used by the client code.
 */
#define CLIENT_2_HDR(a) ((MEMHDR *) (((char *) (a)) - RESERVE_SIZE))
#define HDR_2_CLIENT(a) ((void *) (((char *) (a)) + RESERVE_SIZE))

/* Local variables */
static unsigned long      mem_size = 0 ;  /* Amount of memory used */
#if defined(MEM_LIST)
static MEMHDR     *memlist = NULL ;       /* List of memory blocks */
#endif

/* Local functions */
void      mem_tag_err(void *, char *, int) ;      /* Tag error */
#if defined(MEM_LIST)
void      mem_list_add(MEMHDR *) ;        /* Add block to list */
void      mem_list_delete(MEMHDR *) ;     /* Delete block from list */
#define Mem_Tag_Err(a) mem_tag_err(a,fil,lin)
#else
#define Mem_Tag_Err(a) mem_tag_err(a,__FILE__,__LINE__)
#endif

/**** Functions accessed only through macros ***************************/

/*----------------------------------------------------------------------
 *+
 *  mem_alloc -- Allocate a memory block
 *  Usage:
 *        void *
 *        mem_alloc(
 *        size_t          size
 *        )
 *  Parameters: size  Size of block in bytes to allocate
 *  Return Value: Pointer to allocated memory block, NULL if not
 *    enough memory.
 *  Description: mem_alloc() makes a protected call to malloc()
 *  Notes: Access this routine using the malloc() macro in mshell.h
 *-
 */
void *
mem_alloc(
#if defined(MEM_WHERE)
size_t            size,
char              *fil,
int               lin
#else
size_t            size
#endif
)

{
MEMHDR            *p ;

/* Allocate memory block */
p = malloc(RESERVE_SIZE + size) ;
if (p == NULL)
  {
  return NULL ;
  }

/* Init header */
p->mh_tag = MEMTAG ;
p->mh_size = size ;
mem_size += size ;
#if defined(MEM_WHERE)
p->mh_file = fil ;
p->mh_line = lin ;
#endif

#if defined(MEM_LIST)
mem_list_add(p) ;
#endif

/* Return pointer to client data */
return HDR_2_CLIENT(p) ;
}

/*----------------------------------------------------------------------
 *+
 *  mem_realloc -- Reallocate a memory block
 *  Usage:
 *        void *
 *        mem_realloc(
 *        void            *ptr,
 *        size_t          size
 *        )
 *  Parameters: ptr               Pointer to current block
 *              size              Size to adjust block to
 *  Return Value: Pointer to new memory block; NULL if memory cannot
 *    be reallocated
 *  Description: mem_realloc() makes a protected call to realloc().
 *  Notes: Access this routine using the realloc() macro in mshell.h
 *-
 */

void *
mem_realloc(
#if defined(MEM_WHERE)
void              *ptr,
size_t            size,
char              *fil,
int               lin
#else
void              *ptr,
size_t            size
#endif
)

{
MEMHDR            *p ;

/* Convert client pointer to header pointer */
p = CLIENT_2_HDR(ptr) ;

/* Check for valid block */
if (p->mh_tag != MEMTAG)
  {
  Mem_Tag_Err(p) ;
  return NULL ;
  }

/* Invalidate header */
p->mh_tag = ~MEMTAG ;
mem_size -= p->mh_size ;

#if defined(MEM_LIST)
mem_list_delete(p) ;      /* Remove block from list */
#endif

/* Reallocate memory block */
p = (MEMHDR *) realloc(p, RESERVE_SIZE + size) ;
if (p == NULL)
  {
  return NULL ;
  }

/* Update header */
p->mh_tag = MEMTAG ;
p->mh_size = size ;
mem_size += size ;
#if defined(MEM_WHERE)
p->mh_file = fil ;
p->mh_line = lin ;
#endif

#if defined(MEM_LIST)
mem_list_add(p) ; /* Add block to list */
#endif

/* Return pointer to client data */
return HDR_2_CLIENT(p) ;
}

/*----------------------------------------------------------------------
 *+
 *  mem_strdup -- Save a string in dynamic memory
 *  Usage:
 *        char *
 *        mem_strdup(
 *        char            *str
 *        )
 *  Parameters:  str      String to save
 *  Return Value:  Pointer to allocated string; NULL if not enough memory
 *  Description: mem_strdup() saves specified string in dynamic memory.
 *  Notes: Access this routine using the strdup() macro in mshell.h
 *-
 */

char *
mem_strdup(
#if defined(MEM_WHERE)
char              *str,
char              *fil,
int               lin
#else
char              *str
#endif
)

{
char * s ;

#if defined(MEM_WHERE)
s = mem_alloc(strlen(str)+1, fil, lin) ;
#else
s = mem_alloc(strlen(str)+1) ;
#endif

if (s != NULL)
  {
  strcpy(s, str) ;
  }

return s ;
}

/*----------------------------------------------------------------------
 *+
 *  mem_free -- Free a memory block
 *  Usage:
 *        void
 *        mem_free(
 *        void            *ptr
 *        )
 *  Parameters: ptr       Pointer to memory to free
 *  Return Value: None
 *  Description: mem_free() frees specified memory block. The block must
 *  be allocated using mem_alloc(), mem_realloc() or mem_strdup().
 *  Notes: Access this routine using the free() macro in mshell.h
 *-
 */

void
mem_free(
#if defined(MEM_WHERE)
void              *ptr,
char              *fil,
int               lin
#else
```

(Listing continued on page 114)

Listing Four *(Listing continued, text begins on page 24.)*

```
290: void              *ptr
291: #endif
292: )
293:
294: {
295: MEMHDR             *p ;
296:
297: /* Convert client pointer to header pointer */
298: p = CLIENT_2_HDR(ptr) ;
299:
300: /* Check for valid block */
301: if (p->mh_tag != MEMTAG)
302:   {
303:   Mem_Tag_Err(p) ;
304:   return ;
305:   }
306:
307: /* Invalidate header */
308: p->mh_tag = ~MEMTAG ;
309: mem_size -= p->mh_size ;
310:
311: #if defined(MEM_LIST)
312: mem_list_delete(p) ;      /* Remove block from list */
313: #endif
314:
315: /* Free memory block */
316: free(p) ;
317: }
318:
319: /**** Functions accessed directly ************************************/
320:
321: /*-------------------------------------------------------------------------
322:  *+
323:  *  Mem_Used -- Return amount of memory currently allocated
324:  *  Usage:
325:  *        unsigned long
326:  *        Mem_Used(
327:  *        )
328:  *  Parameters: None.
329:  *  Description: Mem_Used() returns number of bytes currently allocated
330:  *    using the memory management system. Value returned is simply the
331:  *    sum of the size requests to allocation routines; does not reflect
332:  *    any overhead required by the memory management system.
333:  *  Notes: None
334:  *-
335:  */
336:
337: unsigned long
338: Mem_Used(
339: void)
340:
341: {
342: return mem_size ;
343: }
344:
345: /*-------------------------------------------------------------------------
346:  *+
347:  *  Mem_Display -- Display memory allocation list
348:  *  Usage:
349:  *        void
350:  *        Mem_Display(
351:  *        FILE            *fp
352:  *        )
353:  *  Parameters: fp        File to output data to
354:  *  Description: Mem_Display() displays contents of memory allocation
355:  *    list. This function is a no-op if MEM_LIST is not defined.
356:  *  Notes: None
357:  *-
358:  */
359:
360: void
361: Mem_Display(
362: FILE              *fp
363: )
364:
365: {
366: #if defined(MEM_LIST)
367: MEMHDR            *p ;
368: int               idx ;
369:
370: #if defined(MEM_WHERE)
371: fprintf(fp, "Index   Size  File(Line) - total size %lu\n", mem_size) ;
372: #else
373: fprintf(fp, "Index   Size - total size %lu\n", mem_size) ;
374: #endif
375:
376: idx = 0 ;
377: p = memlist ;
378: while (p != NULL)
379:   {
380:   fprintf(fp, "%-5d %6u", idx++, p->mh_size) ;
381: #if defined(MEM_WHERE)
382:   fprintf(fp, "  %s(%d)", p->mh_file, p->mh_line) ;
383: #endif
384:   if (p->mh_tag != MEMTAG)
385:           {
386:           fprintf(fp, " INVALID") ;
387:           }
388:   fprintf(fp, "\n") ;
389:   p = p->mh_next ;
390:   }
391: #else
392: fprintf(fp, "Memory list not compiled (MEM_LIST not defined)\n") ;
393: #endif
394: }
395:
396: /**** Memory list manipulation functions *****************************/
397:
398: /* mem_list_add() -- Add block to list  */
399:
400: #if defined(MEM_LIST)
401: static void
402: mem_list_add(
403: MEMHDR    *p
404: )
405:
406: {
407: p->mh_next = memlist ;
408: p->mh_prev = NULL ;
409: if (memlist != NULL)
410:   {
411:   memlist->mh_prev = p ;
412:   }
413: memlist = p ;
414:
415: #if defined(DEBUG_LIST)
416: printf("mem_list_add()\n") ;
417: Mem_Display(stdout) ;
418: #endif
419: }
420: #endif
421:
422: /*-------------------------------------------------------------------------*/
423: /* mem_list_delete() -- Delete block from list  */
424:
425: #if defined(MEM_LIST)
426: static void
427: mem_list_delete(
428: MEMHDR    *p
429: )
430:
431: {
432: if (p->mh_next != NULL)
433:   {
434:   p->mh_next->mh_prev = p->mh_prev ;
435:   }
436: if (p->mh_prev != NULL)
437:   {
438:   p->mh_prev->mh_next = p->mh_next ;
439:   }
440:   else
441:   {
442:   memlist = p->mh_next ;
443:   }
444:
445: #if defined(DEBUG_LIST)
446: printf("mem_list_delete()\n") ;
447: Mem_Display(stdout) ;
448: #endif
449: }
450: #endif
451:
452: /**** Error display ***************************************************/
453:
454: /* mem_tag_err() -- Display memory tag error  */
455: static void
456: mem_tag_err(
457: void              *p,
458: char              *fil,
459: int               lin
460: )
461:
462: {
463: fprintf(stderr, "Memory tag error - %p - %s(%d)\n", p, fil, lin) ;
464: #if defined(MEM_LIST)
465: Mem_Display(stderr) ;
466: #endif
467: exit(1) ;
468: }
469:
470: /*-------------------------------------------------------------------------*/
```

End Listings

Listing One *(Text begins on page 36.)*

```
#    This program reads a C structure and produces C code
#    to write the record to a C-ISAM work area.
#    There are limitations on the input format:  see sample

BEGIN {     FS = "[ \t[]+"              # Override default
      }                                 #    field separators
$1 == "struct" && $3 == "{" {           # Opening record struct
            rec = $2
            print "/*  " $2 "  */"
            offset = 0
       }
$2 == "struct" && $3 == "{" {           # Struct within record
            type = "struct"
            j = 0
       }
$2 == "long"   {    f(type == "struct", "stlong",  $3, 4)}
$2 == "int"    {    f(type == "struct", "stint",   $3, 2)}
$2 == "float"  {    f(type == "struct", "stfloat", $3, 4)}
$2 == "double" {    f(type == "struct", "stdbl",   $3, 8)}
$2 == "char" && NF > 3 {                # String
                    f(type == "struct", "stchar",  $3, $4 - 1)}
$2 == "char" && NF == 3 {               # Single character
                    f(type == "struct", "stchar",  $3, 1)}
$2 == "}" && NF > 3 {                   # Array of structs
            type = ""
            print "\tfor (i = 0; i < " $4 + 0 "; i++) {"
            for (k = 0; k < j; k++) {
                gsub(";", "", name[k])
                temp = $3 "[i]." name[k]
                emit2("\t" stype[k], temp, flen[k])
            }
            print "\t}"
            offset += ($4 - 1) * slen
            slen = 0
        }

$2 == "}" && NF == 3 {                  # Named struct
            type = ""
            for (k = 0; k < j; k++)
                emit(stype[k], $3 "." name[k], flen[k])
            slen = 0
        }

function f(bool, str, x, y) {
           if (bool)
                setup(str, x, y)
           else
                emit(str, x, y)
        }

function setup(type, varname, l) {      # Save field data in array
            name[j] = varname
            stype[j] = type
            flen[j++] = l
            slen += l
        }

function emit(type, varname, l) {       # Print C code for field
            gsub(";", "", varname)
            print "\t" type "(p." rec "->" varname,
                ", inf_rec + " offset \
                (type == "ststring" ? ", " l ");" : ");")
            offset += l
        }

function emit2(type, varname, l) {      # Print C code for field in struct
            gsub(";", "", varname)
            print "\t" type "(p." rec "->" varname,
                ", inf_rec + i * " slen, "+",
                offset (type ~ /string/ ? ", " l ");": ");")
            offset += l
        }
```

End Listing

Listing One *(Text begins on page 48.)*

```
/* include all of the files that you might need. */

#include <ctype.h>
#include <memory.h>
#include <math.h>
#include <types.h>
#include <quickdraw.h>
#include <palette.h>
#include <toolutils.h>
#include <fonts.h>
#include <events.h>
#include <windows.h>
#include <dialogs.h>
#include <stdio.h>
#include <menus.h>
#include <desk.h>
#include <textedit.h>
#include <scrap.h>
#include <segload.h>
#include <controls.h>
#include <packages.h>
#include <slots.h>
#include <ShutDown.h>
#include <errors.h>
#include <files.h>

/* ------------------------------------------------------------------------ */
/* The low-level help functions for graphics. */

/* Mean-squared distance between to coordinate pairs. */

int distance(i0, j0, i1, j1)
int i0, j0, i1, j1;
{ int res;
  double x1, x2, y1, y2;

  x1 = i0; x2 = i1; y1 = j0; y2 = j1;
  res = sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));
  return(res);
}

/* ------------------------------------------------------------------------ */
/* The low-level SPLINE function. This function calculates and draws a
curved line along the path of a bi-cubic spline the equation for a coordinate
of the points on the spline is:
j = j1*t + j2*(1 - t) + W*(j2-j3)*t*(1 - t)*(1 - t) + W*(j1-j0)*t*t*(1 - t);
   where:       j : the coordinate being calculated.
                t : the cubic variable.
                W : a weighting factor.
                (j1, j2) : the end-coordinates of the spline.
                (j0, j3) : The control coordinates.
*/

_Spline (i0, j0, i1, j1, i2, j2, i3, j3)
int i0, j0, i1, j1, i2, j2, i3, j3;
{ float dt, t0, t1, t2, t3, t5, inc, dist, W;
   int i, j, i5, j5, count;

   inc = 2.5;                        /* Two pixel wide sampling rate */
   W = 0.9;                          /* Set the weighting factor. */
   MoveTo(j1, i1);                   /* Move to P1 */
   dist = distance(i1, j1, i2, j2); /* The distance between the two points*/
   if (inc < dist)
   { j5 = j1-j0;
     i5 = i1-i0;
     j3 = j2-j3;
     i3 = i2-i3;
     dt = inc/dist;     /* Transform the sampling rate to a function of T */
     count = 1.0/dt;    /* Number of samples to be taken in a unit interval */
     while (count-- > 0)
     { t1 = dt*count;        /* calculate I and J as a function of T */
       t2 = 1.0 - t1;                    /*  (1 - t)    */
       t5 = W*t1*t1*t2;                  /*  W*t*t*(1 - t)  */
       t3 = W*t1*t2*t2;                  /*  W*t*(1 - t)*(1 - t) */
       j = j1*t1 + j2*t2 + t3*j3 + j5*t5;
       i = i1*t1 + i2*t2 + t3*i3 + i5*t5;
       LineTo(j, i);
   } }
   else LineTo(i2, j2);
}
```

End Listing One

Listing Two

```
;; ---------------------------------------------------------------------
;; You need to change this to the directory where you have stored FF.fasl
(load "Lauzzana:Ray:Projects:Artifex:LLIB:FF")

;; ---------------------------------------------------------------------
;; You need to change this to the directory where C libraries is stored
(def-logical-pathname "CLIB;" "Lauzzana:Ray:Projects:Artifex:CLIB")

;; ---------------------------------------------------------------------
;; The high-level SPLINE function
(defun SPLINE (&rest p)
"
   SPLINE (&rest point-list)

   Sets up the front window as a graphics port to draw in, and calls
   DO-SPLINE to do the actual work of drawing a spline.

"
  (with-port (ask (front-window) wptr) (do-spline p)))

(defun DO-SPLINE (p)
"
   DO-SPLINE (point-list)

   A smooth curve coonecting the points.

"
(let ((p1 (car p))
         (p2 (cadr p))
         (p3 (caddr p))
         (p4 (cadr (cddr p)))
         (p5 (cadr (cdddr p))))
     (cond ((or (null (pointp p1)) (null (pointp p4))) nil)
           ((and (pointp p1) (pointp p2) (pointp p3) (pointp p4))
              (_Spline (I-coord p1)
                       (J-coord p1)
                       (I-coord p2)
                       (J-coord p2)
                       (I-coord p3)
                       (J-coord p3)
                       (I-coord p4)
                       (J-coord p4))
              (if (pointp p5)
                (append (list 'SPLINE p1) (cdr (DO-SPLINE (cdr p))))
                (cons 'SPLINE p))))))

;; ---------------------------------------------------------------------
;; The high-level help functions for graphics
;; @@, representation for a point.
(defun @@ (a &optional b)
"
  @@ () or (I-coordinate J-coordinate) or (integer)
  A single user point or a point.

  A screen location, in physical coordinates. (@@ 0 0) is the upper-left
  corner. If single integer is recieved it is interpretted as a packed point,
  and it is unpacked into a coordinate pair.

"
  (cond ((and (numberp a) (numberp b)) (list '@@ (round a) (round b)))
        ((numberp a) (let ((i (floor (/ a 65536))))
                       (list '@@  i (- a (* 65536 i)))))))

;; The type test for a POINT.
(defun POINTP (p)
"
  POINTP (item)

  If the item is a point represented in in coordinates, ie. (@@ I J)
  then TRUE, else NIL.

"
  (and (listp p)
       (or (= 3 (length p)) (= 2 (length p))) (equal (car p) '@@)))

;; I-COORDINATE
(defun I-COORD (p)
"
    I-COORD (point)

    Returns the vertical screen coordinate of a point.

"
    (if (pointp p) (cadr p)))

;; J-COORDINATE
(defun J-COORD (p)
"
    J-COORD (point)

    Returns the horizontal screen coordinate of a point.

"
    (if (pointp p) (caddr p)))
;; ---------------------------------------------------------------------
;; The binding to the low-level SPLINE function
;; There are two methods to bind a C function to Allegro.
;; The first method uses DEFFCFUN. Though this is a simpler function,
;; The binding will be lost if you build a stand-alone application.
;; Using DEFFCFUN the code would be:
;; (deffcfun (Mac_Spline "_Spline")
;;  (integer integer integer integer integer integer integer integer)
:novalue)
;; The second and preferred method uses MULTIPLE-VALUE-BIND and FF-CALL
;; to  bind a physical entry-point to a symbol. This is a hardware dependent
;; solution,  in that you need to represent the binding in terms of the
;; physical size and ordering of the machines stack. In the case of the
;; Mac II, parameters are stored in reverse order on the stack. Therefore,
;; the foreign function call  must reverse the order of the parameters.
;; MULTIPLE-VALUE-BIND is used to perform the mapping of parametes.
;; FF-LOOKUP-ENTRY find the physical address of the entry-point for function.
;; FF-CALL executes the coda stored at a physical address.

(defun _Spline (a b c d e f g h)
"
   _Spline (i0 j0 i1 j1 i2 j2 i3 j3)

    Draws a spline between p1 and p2 to the control points p0 and p3.
 "
   (multiple-value-bind (entry a5) (ff-lookup-entry "_Spline")
       (ff-call entry
         :a5 a5 :long h :long g :long f :long e :long d :long c :long b :long a
         :novalue)))

;; ---------------------------------------------------------------------
;; The loading the low-level SPLINE function
;; FF-LOAD is used to load binary files and their associated libraries.
;; In this case the object module for the function _Spline is stored in
;; the file <spline.c.o>. FF-LOAD searches this file for the symbolic name
;; _Spline and load the binary code associated with it.
;; In addition, it searches the libraries for unsatisfied symbolic references
;; which which may have occurred within the code and loads them as well.
;; In otherwords, it links and loads.
;; You need to change this to the directory where you have stored your C
spline

(ff-load "Lauzzana:Ray:Papers:Spline:spline.c.o"
         :entry-names
         (list "_Spline" )
         :libraries
         (list "CLIB;StdCLib.o" "CLIB;CRuntime.o" "CLIB;CInterface.o"
               "CLIB;math.o" "CLIB;CSANElib.o"))
```

End Listings

Listing One *(Text begins on page 60.)*

```
/* mfmt.h -- macros and function prototypes for mfmt routine. */

#define printf   AltPrintf
#define fprintf  AltFprintf
#define sprintf  AltSprintf
#define vprintf  AltVprintf
#define vfprintf AltVfprintf
#define vsprintf AltVsprintf

int AltPrintf   (char *fmt, ...);
int AltFprintf  (FILE *f, char *fmt, ...);
int AltSprintf  (char *Dest, char *fmt, ...);
int AltVprintf  (char *fmt, va_list Arg);
int AltVfprintf (FILE *f, char *fmt, va_list Arg);
int AltVsprintf (char *Dest, char *fmt, va_list Arg);
```

End Listing One

Listing Two

```
/* mfmt.c -- function to output comma-separated numbers with printf().
 * Copyright 1990, Jim Mischel
 * To compile for testing:
 *   tcc -ms -f mfmt
 * To compile for use as a called module:
 *   tcc -ms -c -f mfmt
 */
/* #define TESTING              /* remove comment for testing */
#include "stdio.h"
#include "stdlib.h"
#include "ctype.h"
#include "string.h"
#include "math.h"

#define AddBlockArg(ap, type) (*((type *)(NewArgPtr))++) = va_arg(ap, type)
#define RemoveBlockArg(type) ((type *)(NewArgPtr))--

static va_list OldArgPtr;       /* Pointer to passed arguments */
static va_list NewArgPtr;       /* Pointer to new argument block */
static va_list NewBlock = NULL; /* New argument block */
static char *Sptr;              /* Pointer to passed format string */
static char *Num;               /* Formatted number */
static char *NumPtr;            /* Pointer into formatted number string */
static char *t = NULL;          /* New format string */
unsigned tSize;                 /* Length of new format string */
static char *Tptr;              /* Pointer to new format string */
static char *SavePtr;           /* Saved Tptr used when %m format found */
static int Flags;               /* Flags identified in format string */
static int Width;               /* Width from format string */
static int WidthFlag;           /* Identifies width type */
static int Precision;           /* Width from format string */
static int PrecisionFlag;       /* Identifies precision type */
static int Size;                /* printf() size modifier */
static int Sign;                /* Sign of number for formatting */

/* Increment the flags counter for each occurance of a flag character.
 * Valid flag characters are blank, '-', '#', '+'.  */
/* Definitions for flag characters */
#define Blank 1
#define Dash 2
#define Pound 4
#define Plus 8

static void ParseFlags (void) {
    Flags = 0;
    while (*Sptr == ' ' || *Sptr == '-' || *Sptr == '#' || *Sptr == '+')
        switch (*Tptr++ = *Sptr++) {
            case ' ' : Flags += Blank; break;
            case '-' : Flags += Dash; break;
            case '#' : Flags += Pound; break;
            case '+' : Flags += Plus; break;
        } /* switch */
} /* ParseFlags */

/* Width specification.  Minimum field width is returned in the Width variable.
 * WidthFlag specifies if '0' or '*' was given in the format string.  */
static void ParseWidth (void) {
    Width = WidthFlag = 0;

    if (*Sptr == '0')
        WidthFlag = (*Tptr++ = *Sptr++);
    if (*Sptr == '*') {
        WidthFlag |= 0x80;
        *Tptr++ = *Sptr++;
        Width = (AddBlockArg(OldArgPtr, int));
        if (Width < 0) {
            Width = abs (Width);
            Flags |= Dash;
        }
    }
    while (isdigit (*Sptr)) {
        Width = Width * 10 + *Sptr - '0';
        *Tptr++ = *Sptr++;
    }
} /* ParseWidth */

/* Precision specification. Returns precision in the variables Precision and
 * PrecisionFlag.
 * PrecisionFlag = 0, no precision was specified
 * PrecisionFlag = '0', ".0" specified
 * PrecisionFlag = 'n', ".n" specified
 * PrecisionFlag = '*', ".*" specified
 */
static void ParsePrecision (void) {
    Precision = PrecisionFlag = 0;

    if (*Sptr == '.') {         /* precision specified */
```

```
        *Tptr++ = *Sptr++;
        if (*Sptr == '*') {
            PrecisionFlag = (*Tptr++ = *Sptr++);
            Precision = (AddBlockArg(OldArgPtr, int));
        }
        else if (*Sptr == '0')
            PrecisionFlag = (*Tptr++ = *Sptr++);
        else {
            PrecisionFlag = 'n';
            while (isdigit (*Sptr)) {
                Precision = Precision * 10 + *Sptr - '0';
                *Tptr++ = *Sptr++;
            }
        }
    }
} /* ParsePrecision */

/* Input size modifier.  (F|N|h|l|L) */
static void ParseSize (void) {
    if (*Sptr == 'F' || *Sptr == 'l' || *Sptr == 'L' ||
        *Sptr == 'N' || *Sptr == 'h')
        Size = (*Tptr++ = *Sptr++);
    else
        Size = 0;
} /* ParseSize */

/* dtoa -- convert a double to comma-separated ASCII representation.  */
static void dtoa (double Work, int P, int Digits) {
    int c;

    if (P >= 0 || Work != 0) {
        if (P == 0) {
            c = '.';
            Digits = 0;
            P--;
        }
        else if (Digits == 3) {
            c = ',';
            Digits = 0;
        }
        else {
            c = (int)(fmod (Work, 10))+'0';
            modf (Work/10, &Work);
            if (P > 0)
                P--;
            else
                Digits++;
        }
        dtoa (Work, P, Digits);
        *NumPtr++ = c;
    }
} /* dtoa */

/* Right- or left- justifies the formatted number in the string.
 * If the number is too large for the specified width, the number
 * field is filled with the asterisk character (*). */
static void Justify (void) {
    if ((WidthFlag & 0x7f) == '0')      /* check width */
        if (strlen (Num) > Width) {
            memset (Num, '*', Width);
            NumPtr = Num + Width;
            *NumPtr = '\0';
            return;
        }
    if (strlen (NumPtr) < Width) {
        if (Flags & Dash)       /* left justify */
            memset (NumPtr, ' ', Width - strlen (Num));
        else {                  /* right justify */
            int n = Width - strlen (Num);
            char *p = Num;
            memmove (p + n, p, strlen (Num));
            memset (p, ' ', n);
        }
        NumPtr = Num + Width;
        *NumPtr = '\0';
    }
} /* Justify */

/* Reports an out of memory error and aborts the program. */
static void MemoryError (char *s) {
    fprintf (stderr, "Out of memory in function %s\n", s);
    exit (1);
} /* MemoryError */

/* Format 'm' type into the new format string t. All parameters (flags, width,
 * precision, size, and type) are stored in the respective variables.  */
static void mFormat (void) {
    double Work;
    /* Move the block pointer back where it belongs if there were width or
     * precision specifications in the argument list. */
    *SavePtr = '\0';
    if (PrecisionFlag == '*')
        RemoveBlockArg (int);
    if (WidthFlag & 0x80)
        RemoveBlockArg (int);

    if ((Num = malloc (128)) == NULL)
        MemoryError ("mFormat");
    NumPtr = Num;
    /* The next argument in the list is the number that is to be formatted.
     * This number will either be a float, a double, or a long double. The
     * input size modifier will tell us what type. Whatever type it is, it will
     * be copied into the double variable Work for us to work with. */
    if (Size == 'l')      Work = va_arg (OldArgPtr, double);
    else if (Size == 'L') Work = (double) va_arg (OldArgPtr, long double);
    else                  Work = (double) va_arg (OldArgPtr, float);
    Sign = (Work < 0) ? -1 : 1;
    if (!PrecisionFlag)
        Precision = 2;
```

(Listing continued on page 122)

Listing Two *(Listing continued, text begins on page 60.)*

```
    dtoa (floor (fabs (Work) * pow10 (Precision)),
          (Precision == 0) ? -1 : Precision, 0);
    *NumPtr = '\0';
    /* The number is formatted into Num. Precision was handled by the dtoa()
     * function. Add the sign and perform padding/justifying as necessary.
     * Determine the proper sign */
    if (Sign == -1)         Sign = '-';
    else if (Flags & Plus)  Sign = '+';
    else if (Flags & Blank) Sign = ' ';
    else                    Sign = '\0';

    if (Sign != '\0')           /* Place sign */
        if (Flags & Pound) {    /* trailing sign */
            *NumPtr++ = Sign;
            *NumPtr = '\0';
        }
        else {                  /* leading sign */
            memmove (Num+1, Num, (NumPtr - Num));
            *Num = Sign;
            NumPtr++;
        }
    Justify ();
    /* Now re-allocate the string to add more characters and then append the
     * newly-formatted number to the string and release the memory taken by
     * the formatted number string. */
    if ((t = realloc (t, (tSize += strlen (Num)))) == NULL)
        MemoryError ("mFormat");
    strcat (t, Num);
    Tptr = t + strlen (t);
    free (Num);
} /* mFormat */

/* Determine the conversion type. For any type but 'm', simply copy the passed
 * argument to the new argument list and return.  */
static void ParseType (void) {
    switch (*Tptr++ = *Sptr++) {
        case 'd' :
        case 'i' :
        case 'o' :
        case 'u' :
        case 'x' :
        case 'X' :
        case 'c' :              /* in Turbo C, char is treated as int */
            if (Size == 'l') AddBlockArg(OldArgPtr, long);
            else             AddBlockArg(OldArgPtr, int);
            break;
        case 'f' :
        case 'e' :
        case 'g' :
        case 'E' :
        case 'G' :
            if (Size == 'l')      AddBlockArg(OldArgPtr, double);
            else if (Size == 'L') AddBlockArg(OldArgPtr, long double);
            else                  AddBlockArg(OldArgPtr, float);
            break;
        /* In Turbo C, pointers to all types are the same size */
        case 's' :
        case 'n' :
        case 'p' :
            if (Size == 'F')      AddBlockArg(OldArgPtr, char far *);
            else if (Size == 'N') AddBlockArg(OldArgPtr, char near *);
            else                  AddBlockArg(OldArgPtr, char *);
            break;
        case 'm' :
            mFormat ();         /* format 'm' type */
            break;
        default  :              /* anything else isn't defined */
            break;
    } /* switch */
} /* ParseType */

/* Parse a standard printf() format.  */
static void ParseFormat (void) {

    SavePtr = Tptr - 1;

    ParseFlags ();
    ParseWidth ();
    ParsePrecision ();
    ParseSize ();
    ParseType ();
} /* ParseFormat */

/* Copy the input format string to the new format string t, replacing all %m
 * formats with the formatted digit string. Arguments will be placed in the
 * ParamBlock structure, with the %m parameters removed. */
static void Formatter (char *s, va_list Arg) {
    Sptr = s;
    /* Allocate memory for new format string. */
    if ((t = malloc (tSize = strlen (s))) == NULL)
        MemoryError ("Formatter");
    Tptr = t;
    OldArgPtr = Arg;
    while (*Sptr != '\0')
        if ((*Tptr++ = *Sptr++) == '%')
            ParseFormat ();
    va_end (OldArgPtr);
    *Tptr = '\0';
} /* Formatter */

/* Allocate a block of memory for the modified argument list.  */
static void InitBlock (char *s) {
    int BlockSize = 0;

    while (*s != '\0')          /* count the '%' characters */
        if (*s == '%')          /* in the format string */
            BlockSize++;
```

```
    /* Multiply the number of '%' by the size of a long double, giving some
     * idea of the maximum required size of the new parameter block. It's
     * crude and implementation dependent, but it works. */
    BlockSize *= sizeof (long double);
    if ((NewBlock = malloc (BlockSize)) == NULL)
        MemoryError ("InitBlock");
    NewArgPtr = NewBlock;
} /* InitBlock */

/* Free the memory used by the modified argument list (if any) and the
 * modified format string.  */
static void FreeBlock (void) {
    free (t);
    free (NewBlock);
    NewBlock = t = NULL;
} /* FreeBlock */

/* These 6 routines are the only functions visible to the application program.
 * They are accessed though the printf, fprintf, sprintf, vprintf, vfprintf,
 * and vsprintf macros, respectively.  */
int AltPrintf (char *fmt, ...) {
    va_list Arg;
    int r;
    va_start (Arg, fmt);
    Formatter (fmt, NewArgPtr = Arg);
    r = vprintf (t, Arg);
    FreeBlock ();
    return r;
} /* AltPrintf */

int AltFprintf (FILE *f, char *fmt, ...) {
    va_list Arg;
    int r;
    va_start (Arg, fmt);
    Formatter (fmt, NewArgPtr = Arg);
    r = vfprintf (f, t, Arg);
    FreeBlock ();
    return r;
} /* AltFprintf */

int AltSprintf (char *Dest, char *fmt, ...) {
    va_list Arg;
    int r;
    va_start (Arg, fmt);
    Formatter (fmt, NewArgPtr = Arg);
    r = vsprintf (Dest, t, Arg);
    FreeBlock ();
    return r;
} /* AltSprintf */

int AltVprintf (char *fmt, va_list Arg) {
    int r;
    InitBlock (fmt);
    Formatter (fmt, Arg);
    r = vprintf (t, NewBlock);
    FreeBlock ();
    return r;
} /* AltVprintf */

int AltVfprintf (FILE *f, char *fmt, va_list Arg) {
    int r;
    InitBlock (fmt);
    Formatter (fmt, Arg);
    r = vfprintf (f, t, NewBlock);
    FreeBlock ();
    return r;
} /* AltVfprintf */

int AltVsprintf (char *Dest, char *fmt, va_list Arg) {
    int r;
    InitBlock (fmt);
    Formatter (fmt, Arg);
    r = vsprintf (Dest, t, NewBlock);
    FreeBlock ();
    return r;
} /* AltVsprintf */

#ifdef TESTING
#include "mfmt.h"
void main (void) {
    printf ("The national debt exceeds $%.0lm\n", (double)1000000000000.0);
 }
 #endif
```

End Listings

Listing One *(Text begins on page 70.)*

```
/**** File: proto.h -- Contains the function protocols ****/
/* include mytypes.h first....I don't do any checking */

source(Process *, Channel *, UBYTE *);
output(Process *, Channel *);
pipe(Process *, Channel *, Channel *);

/* end proto.h */
```

End Listing One

Listing Two

```
/**** mytypes.h ****/

#define BUFF_SIZE       256         /* the max. char buffer size */
#define NUM_PIPES       BUFF_SIZE   /* the number of sorting processes */
#define PIPE_STACK      128         /* stack space for the processes*/
#define INPUT_STACK     4096
#define OUTPUT_STACK    4096

#define TRUE            1           /* some simple defines */
#define FALSE           0

typedef int                             BOOL;
typedef unsigned char   UBYTE;
```

End Listing Two

Listing Three

```
;  multisort.nif -- This is the file the transputer uses to load the network.
;  The parameters are:
;-------------------------------------------------------------------------
; node number   ¦   processor reset ¦ link 0 ¦ link1 ¦ link 2¦ link 3 ¦
;               ¦   from processor  ¦---------------------------------¦
;               ¦   number Rxx      ¦ The number here is the processor¦
;               ¦                   ¦ number we connect to.           ¦
;-------------------------------------------------------------------------
;
1, multisort, R0, 0, , , 2;
2, multisort, R1,  , ,1,  ;
```

End Listing Three

Listing Four

```
/**** File: buffers.c -- Contains the funtions:
**      source(Process *, Channel *, UBYTE *)
**      output(Process *, Channel *)
**      pipe(Process *, Channel *, Channel *)
****/

#include <stdio.h>
#include <conc.h>
#include "mytypes.h"            /* include in this order...     */
#include "proto.h"

/**** source(Process *p, Channel *out, UBYTE *ch)
**    Output a single character at a time from the buffer pointed to by ch
**    on the channel out.  Terminate when '\0' or whole buffer is sent.
**    Process *p is used by the ProcAlloc() routine. You must have this
**    in the parameter list for a parallel process.
****/
source(Process *p, Channel *out, UBYTE *ch)
{
        BOOL inline = TRUE;
        int i;
        for(i=0; i<BUFF_SIZE; i++) {
                if(inline)
                        switch(ch[i]) {
                                case '\0':   /* this is the EOL for us */
                                        ChanOutChar(out, ch[i]);
                                        inline = FALSE;
                                        break;
                                default:
                                        ChanOutChar(out, ch[i]);
                        }
        }
}

/**** output(Process *p, Channel *in)
**   Read a single character at at time on channel in and send it to stdout.
**   Terminate when '\0' is received. Process *p is used by the ProcAlloc()
**   routine. You must have this in the parameter list for a parallel process.
****/
output(Process *p, Channel *in)
{
        UBYTE ch;
        BOOL running = TRUE;

        while(running) {
                ch = ChanInChar(in);
                switch(ch) {
                        case '\0':
                                putchar('\n');
                                running = FALSE;
                                break;
                        case '\n':
                                break;
                        default:
                                putchar(ch);
                }
        }
}

/**** pipe(Process *p, Channel *in, Channel *out)
**  Reads a single character on channel in and falls into a loop where next
**  character is read and compared in an ASCII sense to the current charater.
**  Character with the lowest ASCII value is forwarded on channel out. Loop
**  terminates on a '\0' character.  The "stored" character is sent before the
**  '\0' is propagated. Process *p is used by the ProcAlloc() routine.
**  You must have this in the parameter list for a parallel process.
****/
pipe(Process *p, Channel *in, Channel *out)
{
        UBYTE highest, next;
        BOOL running;
        BOOL ns_node;

        if(_node_number == 1)    /* If we are the root node only do the */
                ns_node = FALSE; /* main loop once.  If we are a non-   */
        else                     /* root node, loop forever.            */
                ns_node = TRUE;
        do {
                running = TRUE;
                highest = ChanInChar(in);
                while(running) {
                        next = ChanInChar(in);
                        switch(next) {
                                case '\0' :
                                        ChanOutChar(out, highest);
                                        running = FALSE;
                                        break;
                                default :
                                        if(next > highest) {
                                                ChanOutChar(out, highest);
                                                highest = next;
                                        }
                                        else
                                        {
                                                ChanOutChar(out, next);
                                        }
                        }
                }
                ChanOutChar(out, '\0');
        } while(ns_node);
}
```

End Listing Four

Listing Five

```
/*************************************************************************
**  sort.c -- Compiles under Logical Systems Transputer C compiler Versions
**  88.4, 89.1. A single transputer, multi-process ASCII sorting routine. This
**  shows how to allocate and deallocate parallel (multi-tasking) processes on
**  a single transputer. This algorithm is modelled after the sorting example
**  in the INMOS occam Transputer Development System (TDS). The algorithm
**  relies on having as many parallel processes as the maximum character
**  buffer size. Data is sent to a FIFO-like array of input-ouput buffers. If
**  the next letter read in is less than (in an ASCII sense) the current letter
**  in the buffer, it is forwarded to the next buffer, otherwise the current
**  data is forwarded. The program will shut down when a ~ is the first
**  character in any input line.
**  Must link in file buffers.c (after compilation of course..)
**  Programed by: G.K.Ellis, Smart Materials and Structures Laboratory,
**  Mechanical Engineering Department, Virginia Tech, Blacksburg, VA 24061
*************************************************************************/

#include <stdio.h>
#include <conc.h>               /* the LSC defines are here */
#include "mytypes.h"            /* include in this order... */
#include "proto.h"

/**** MAIN PROGRAM  ****/
void main()
{
        UBYTE ch[BUFF_SIZE];     /* string buffer */
        BOOL active = TRUE;
        int i;
        Channel *thru[NUM_PIPES + 1];   /* internal channels */
        Process *p[NUM_PIPES + 3];      /* process pointers */
        while(active) {
                fgets(ch, BUFF_SIZE-1, stdin);  /* get the input line */
                /* stop char is ~ as first letter */
                if(ch[0] == '~') active = 0;
                /* allocate the channels, NULL is returned if no memory */
                for(i = 0; i< NUM_PIPES + 1; i++) {
                        if((thru[i] = ChanAlloc()) == NULL)
                                printf("No memory for Channel thru[%d]\n", i);
                }

        /* allocate the processes, NULL is returned if no memory */
        p[0] = ProcAlloc(source, INPUT_STACK, 2, thru[0], ch);
        for(i = 1; i < NUM_PIPES+1; i++)
            p[i] = ProcAlloc(pipe, PIPE_STACK, 2, thru[i-1], thru[i]);
        p[NUM_PIPES+1] = ProcAlloc(output, OUTPUT_STACK, 1,  thru[NUM_PIPES]);
            p[NUM_PIPES+2] = NULL;
        /* check to see if all processes were allocated successfully    */
                for(i = 0; i < NUM_PIPES + 2; i++)
                        if(!p[i])
                                printf("No memory for Process p[%d]\n", i);
          ProcParList(p);       /* Run a NULL terminated list of pointers */
                                /* to processes. These are all low-pri  */
        /* Since we allocate these each time through the loop, we needto
```

(Listing continued on page 126)

Listing Five *(Listing continued, text begins on page 70.)*

```
        ** deallocate them here otherwise, we will run out of memory */
                for(i = 0; i < NUM_PIPES + 1; i++)
                        ChanFree(thru[i]);
                for(i = 0; i < NUM_PIPES + 2; i++)
                        ProcFree(p[i]);
        }
        printf("done!\n");              /* all done...  */
```

End Listing Five

Listing Six

```
/*************************************************************************
** multisort.c -- A two transputer version of a simple parallel ASCII sorting
** routine. This program will work on either node. Developed for Logical
** Systems Transputer C compiler (LSC) versions 88.4 or 89.1. Must link with
** buffers.c. Programmed by: G.K.Ellis, Smart Materials and Structures
** Laboratory, Mechanical Engineering Department, Virginia Tech,
** Blacksburg, VA 24061
*************************************************************************/

#include <stdio.h>
#include <conc.h>
#include "mytypes.h"
#include "proto.h"

/**** MAIN PROGRAM ****/
void main()
{
        UBYTE ch[BUFF_SIZE];    /* string buffer */
        BOOL active = TRUE;
        int i;
        Channel *in, *out;              /* link channels */
        Process *feeder, *sink;         /* hi-priority processes */
        Channel *thru[NUM_PIPES - 1];   /* internal soft channels */
        Process *p[NUM_PIPES - 2];      /* low-priority processes */
        if(_node_number == 1) {         /* We are the root node */
                in  = LINK3IN;          /* these are our links */
                out = LINK3OUT;
                while(active) {
                   fgets(ch, BUFF_SIZE-1, stdin);   /* get the input line */
                        /* stop char is ~ as first letter */
                        if(ch[0] == '~') active = 0;
                        /* set up the processes */
                        feeder = ProcAlloc(source, INPUT_STACK, 2, out, ch);
                        sink   = ProcAlloc(output, OUTPUT_STACK, 1,  in);
                        /* Check that ProcAlloc() doesn't return a NULL, but
                        ** since we KNOW from the example sort.c this is ok */

                        /* run the feeder and sink processes in parallel */
                        ProcPar(feeder, sink, NULL);

                        /* free the previously allocated processes */
                        /* note we do this each time through the loop */
                        /* hey, it's only an example */
                        ProcFree(feeder);
                        ProcFree(sink);
                }
                printf("done!\n");
                exit(0);                                /* finis! */
        } else {                                        /* if we are the non-root */
           in = LINK2IN;                                /* node, run this as main() */
           out = LINK2OUT;                              /* these are our links */
           for(i = 0; i < NUM_PIPES - 1; i ++) { /* allocate soft channels */
                        thru[i] = ChanAlloc();
                }
               /* allocate soft channel pipe processes */
               for(i = 0; i < NUM_PIPES - 2; i++)
                     p[i] = ProcAlloc(pipe, PIPE_STACK, 2, thru[i], thru[i+1]);
               /* allocate the processes with the links */
               feeder = ProcAlloc(pipe, PIPE_STACK, 2, in, thru[0]);
               sink   = ProcAlloc(pipe, PIPE_STACK, 2, thru[NUM_PIPES-2], out);
                /* If we want to check the pointer returned from ProcAlloc(),
                ** that is extra programming on a non-root node. Therefore,
                ** don't try to printf() from a non-root node.  There is a
                ** non-server node _ns_printf() that does simple ASCII
                ** transfers out a channel (link), but this generally requires
                ** extra effort to communicate back to the server */

                /* Let's allocate these asynchronously */
                ProcRunHigh(sink);   /* run the hardware links at high pri */
                ProcRunHigh(feeder);
                for(i = 0; i < NUM_PIPES - 2; i++)
                        ProcRunLow(p[i]); /* and soft channels at low pri */
                _ns_exit();     /* we must exit like this on non-server*/
                                /* node or we will exit from the */
                                /* main() program */
          /* ProcRunHigh() and ProcRunLow() spawn new processes asyncronously
          ** from main().  The _ns_exit() does a STOPP on the main process
          ** leaving the two spawned processes. The exit() routine trys
          ** to send a message back to the server running on the PC.  If it
          ** isn't there, i.e. you're not the root, you've got problems.
          ** Note also, if we are not the root node, we never terminate
          ** and restart the pipe processes... we just run the processes ! */
        }
}
```

End Listings

Listing One *(Text begins on page 84.)*

```
**********************
*** Microsoft C 6.0 ***
**********************

#include "stdio.h"

/* prototypes */

void doit(int i);

void (* func_ptr)(int i) = doit;

void doit(int i)
    {
    --> push bp
    --> mov  bp,sp
    --> push di
    --> push si
    --> mov  di,WORD PTR [bp+4]

    int loop;

    for (; i > 0; --i)

    --> or  di,di
    --> jle $EX225

        {
        for (loop = 0; loop < 26; ++loop)

        --> $F227:
        --> sub si,si
        --> mov WORD PTR [bp+4],di

            {
            printf("loop character = %c\n", 0x41 + loop);

            --> $F230:
            --> lea  ax,WORD PTR [si+65]
            --> push ax
            --> mov  ax,OFFSET DGROUP:$SG233
            --> push ax
            --> call _printf
            --> add  sp,4
            --> inc  si
            --> cmp  si,26
            --> jl   $F230
            }

        printf("i / 16 = %d\n\n",i / 16);

        --> mov  ax,di
        --> cwd
        --> xor  ax,dx
        --> sub  ax,dx
        --> mov  cx,4
        --> sar  ax,cl
        --> xor  ax,dx
        --> sub  ax,dx
        --> push ax
        --> mov  ax,OFFSET DGROUP:$SG234
        --> push ax
        --> call _printf
        --> add  sp,4
        --> dec  di
        --> jne  $F227

        }

    --> $EX225:
    --> pop si
    --> pop di
    --> mov sp,bp
    --> pop bp
    --> ret
    --> nop
    }

int main(void)
    {
    func_ptr(100);

    --> mov  ax,100
    --> push ax
    --> call WORD PTR _func_ptr
    --> add  sp,2

    return 0;

    --> sub ax,ax
    --> ret
    }
```

End Listing One

Listing Two

```
****************************************
*** Microsoft C 6.0 (using _fastcall) ***
****************************************

#include "stdio.h"

/* prototypes */

void doit(int i);

void (* func_ptr)(int i) = doit;
```

```
void doit(int i)
    {

    --> push bp
    --> mov  bp,sp
    --> sub  sp,2
    --> push ax
    --> push si

    int loop;

    for (; i > 0; --i)

    --> or  ax,ax
    --> jle $EX225

        {
        for (loop = 0; loop < 26; ++loop)

        --> $F227:
        --> sub si,si

            {
            printf("loop character = %c\n", 0x41 + loop);

            --> $F230:
            --> lea  ax,WORD PTR [si+65]
            --> push ax
            --> mov  ax,OFFSET DGROUP:$SG233
            --> push ax
            --> call _printf
            --> add  sp,4
            --> inc  si
            --> cmp  si,26
            --> jl   $F230

            }

        printf("i / 16 = %d\n\n",i / 16);

        --> mov  ax,WORD PTR [bp-4]
        --> cwd
        --> xor  ax,dx
        --> sub  ax,dx
        --> mov  cx,4
        --> sar  ax,cl
        --> xor  ax,dx
        --> sub  ax,dx
        --> push ax
        --> mov  ax,OFFSET DGROUP:$SG234
        --> push ax
        --> call _printf
        --> add  sp,4
        --> dec  WORD PTR [bp-4]
        --> jne  $F227

        }

        --> $EX225:
        --> pop si
        --> mov sp,bp
        --> pop bp
        --> ret
    }

int main(void)
    {
    func_ptr(100);

    --> mov  ax,100
    --> call WORD PTR _func_ptr

    return 0;

    --> sub ax,ax
    --> ret

    }
```

End Listing Two

Listing Three

```
************************
*** Microsoft C 5.10 ***
************************

#include "stdio.h"

/* prototypes */

void doit(int i);
void (* func_ptr)(int i) = doit;

void doit(int i)
    {
    --> push    bp
    --> mov     bp,sp
    --> sub     sp,2
    --> push    di
    --> push    si

    int loop;

    for (; i > 0; --i)
```

(Listing continued on page 130)

Listing Three *(Listing continued, text begins on page 84.)*

```
    --> cmp WORD PTR [bp+4],0
    --> jle $FB202
    --> mov di,WORD PTR [bp+4]

        {
        for (loop = 0; loop < 26; ++loop)

        --> $L20002:
        --> sub si,si

            {
            printf("loop character = %c\n", 0x41 + loop);

            --> $L20000:
            --> lea  ax,WORD PTR [si+65]
            --> push ax
            --> mov  ax,OFFSET DGROUP:$SG206
            --> push ax
            --> call _printf
            --> add  sp,4
        }

        --> inc si
        --> cmp si,26
        --> jl  $L20000
        --> mov WORD PTR [bp-2],si  ;loop

        printf("i / 16 = %d\n\n",i / 16);

        --> mov  ax,di
        --> cwd
        --> xor  ax,dx
        --> sub  ax,dx
        --> mov  cx,4
        --> sar  ax,cl
        --> xor  ax,dx
        --> sub  ax,dx
        --> push ax
        --> mov  ax,OFFSET DGROUP:$SG207
        --> push ax
        --> call _printf
        --> add  sp,4

        }

        --> dec di
        --> jne $L20002
        --> mov WORD PTR [bp+4],di

        --> $FB202:
        --> pop si
        --> pop di
        --> mov sp,bp
        --> pop bp
        --> ret
        --> nop

    }

int main(void)
    {
    func_ptr(100);

    --> mov  ax,100
    --> push ax
    --> call WORD PTR _func_ptr
    --> add  sp,2

    return 0;

    --> sub  ax,ax
    --> ret
    }
```

End Listing Three

Listing Four

```
*******************
*** Watcom C 7.0 ***
*******************

#include "stdio.h"

/* prototypes */

void doit(int i);
void (* func_ptr)(int i) = doit;

void doit(int i)
    {
    int loop;

    --> push    bx
    --> push    cx
    --> push    dx
    --> mov     cx,ax
    --> jmp     short L3

    for (; i > 0; --i)
        {
        for (loop = 0; loop < 26; ++loop)
            {
            --> L1:
            --> mov     bx,0041H

            printf("loop character = %c\n", 0x41 + loop);

            --> L2:
            --> push    bx
```

```
            --> mov     ax,offset DGROUP:L4
            --> push    ax
            --> call    near ptr printf_
            --> add     sp,0004H
            --> inc     bx
            --> cmp     bx,005bH
            --> jne     L2

            }

        printf("i / 16 = %d\n\n",i / 16);

        --> mov     bx,0010H
        --> mov     ax,cx
        --> cwd
        --> idiv    bx
        --> push    ax
        --> mov     ax,offset DGROUP:L5
        --> push    ax
        --> call    near ptr printf_
        --> add     sp,0004H
        --> dec     cx

        }
    --> L3:
    --> test    cx,cx
    --> jg      L1

    --> pop     dx
    --> pop     cx
    --> pop     bx
    --> ret

    }

int main(void)
    {

    func_ptr(100);

    --> mov     ax,0064H
    --> call    word ptr _func_ptr

    return 0;

    --> xor     ax,ax
    --> ret

    }
```

End Listing Four

Listing Five

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>

#define MAX_TAG 2000
unsigned long get_size(void);
_segment segvar;          /* name a segment for use with based pointers */

/* set up structures and tags within segment segvar */

typedef mytag {
char filename[14];
unsigned long size;
mytag _based(segvar) *next;
} _based(segvar) *PTAG, TAG;

main() {

   PTAG head, curptr;

/* Allocate a based heap of MAX_TAG structs. Put segment address in segvar. */

   if((segvar = _bheapseg(sizeof(TAG) * MAX_TAG))) == NULLSEG){
      printf("error allocating based heap \n");
      exit(-1);
   }

/* Allocate memory within segvar for first structure in linked list */

   if((head = _bmalloc(segvar, sizeof(TAG)) == _NULLOFF) {
   printf("error allocating TAG \n");
   exit(-1);
   }
   head->size = get_size();
   _fstrcpy((char far *) head->filename, get_name()); /* get a
   filename and copy it to segvar */

   if((head->next = _bmalloc(segvar, sizeof(TAG)) == _NULLOFF) {
   printf("error allocating TAG \n");
   exit(-1);
   }
.
.
.
}
unsigned long get_size(void) {
return 1;
}

char *get_name(void) {
return("foo");
}
```

End Listings

Listing One *(Text begins on page 94.)*

```
// COLLECT.CPP : collection example, with multiple inheritance and
// pointers to members.
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

struct item {
  virtual ~item() {} // so collection can properly destroy what it holds
};

// Suppose we have a pre-existing class to hold items:
class collection {
  // in 2.0, "holder" is a global name.  In 2.1, it's hidden:
  struct holder {
    holder * next;  // link to next holder
    item * data; // pointer to actual data
    holder(holder * nxt, item * dat)  // constructor
      : next(nxt), data(dat) {}
  } *head, *cursor;
public:
  // initialize an empty list:
  collection() : head((holder *)NULL), cursor((holder *)NULL) {}
  // clean up the list by removing all the elements:
  ~collection() {
    cursor = head;
    while(cursor) {  // while the list is not empty ...
      delete cursor->data;  // delete the current data
      head = cursor->next;  // head keeps track of where the next holder is
      delete cursor;  // delete the current holder
      cursor = head; // move to the next holder
    }
  }
  // Paste a new item in at the top (this is tricky):
  void add(item * i) { head = new holder(head, i); }
  // reset() and next() return the current item or null, for the list's end:
  item * reset() { // go back to the top of the list
    cursor = head;
    // the list may be empty; only return data if cursor isn't null
    return cursor ? cursor->data : (item *)NULL;
  }
  item * next() {
    // Only move forward if cursor isn't null:
    if(cursor) cursor = cursor->next;
    // only return data if cursor isn't null:
    return cursor ? cursor->data : (item *)NULL;
  }
};

// Now suppose we have a second pre-exising class to hold words,
// keep track of word counts, and print itself in 2 ways:
class word {
  char * w;
  int count;
public:
  word(char * wd) : count(1) {
    w = new char[strlen(wd) + 1];  // allocate space for word
    strcpy(w,wd);  // copy it in
  }
  ~word() {
    printf("%s : %d occurrences\n", w, count);
    delete w;  // free space for word
  }
  int compare(char * testword) {
    int match = !strcmp(w, testword);
    if(match) count++; // count testword if it matches
    return match;
  }
  void print1() { printf("%s\n", w); } // 1 per line
  void print2();
};

// Zortech 2.06 and cfront 2.0 wouldn't allow the following
// function as an inline; Turbo C++ would.
void word::print2() {  // print several words per line
  static plcnt;  // count words on a line
  const words_per_line = 7;
  printf("%s ", w);
  if(++plcnt % words_per_line) return;
  putchar('\n'); // only when remainder is 0
}

// What if we want to make a collection of words?  Multiple
// Inheritance to the rescue:
class worditem : public item, public word {
public:
  worditem(char * wrd) : word(wrd) {}
};

// now we can create a collection of worditems.  Here's an array
// of words to put in our collection:
char * words[] = { "this", "is", "a", "test", "of", "worditem" };

#ifdef TEST1
main() {
  collection c;
  for(int i = 0; i < sizeof(words)/sizeof(words[0]); i++)
    c.add(new worditem(words[i]));
  // NOTE: Zortech C++ 2.06 doesn't work here.
}
#endif // TEST1

// But now we want to count instances of words.  We need to modify the
// collection class so it conditionally adds a word, or just counts it
// if it already exists:

class wordcounter : public collection {
public:
  // Customize for worditems (no overhead):
  void add(worditem * wi) { collection::add(wi); }
  worditem * reset() { return (worditem *)collection::reset(); }
  worditem * next() { return (worditem *)collection::next(); }
  void add_or_count(char * newword) {
    worditem * cur = reset();
    while(cur) {
      // if found, increment the count and quit the search:
      if(cur->compare(newword)) return;
      cur = next();
    }
    // at this point, we didn't find it, so add it to the list:
    add(new worditem(newword));
  }
  // Pointers to members (Zortech 2.06 doesn't support this):
  void apply(void (word::*pmf)()) {
    worditem * wit = reset();
    while(wit) {  // do while list is not empty
      (wit->*pmf)();  // dereference member function pointer
      wit = next();  // get next list element
    }
  }
};

char * words2[] = { "this", "this", "is", "a", "test", "test", "test" };

#ifdef TEST2
main() {
  wordcounter wc;
  for(int i = 0; i < sizeof(words2)/sizeof(words2[0]); i++)
    wc.add_or_count(words2[i]);
  // Now "apply" two different functions to the list:
  wc.apply(&word::print1);
  wc.apply(&word::print2); putchar('\n');
}
#endif // TEST2

// Now, for fun, let's use this class to count the keywords and
// identifiers in a C++ program.  Try this program on itself:
//    collect < collect.cpp > count
// Look up strstr(), strchr() and strtok() in your ANSI C
// library guide.

const char * delimiters = " \t#/(){}[]<>.,;:*+-~!%^&=\\|?\'\"";
const char * digits = "0123456789.";

#ifdef TEST3
main() { // use i/o redirection on the command line.
  wordcounter symbol_table;
  char buf[120];
  while (gets(buf)) { // get from standard input
    if(*buf == '#') continue;  // ignore preprocessor lines
    // strip all quoted strings in the line:
    char * quote = strchr(buf, '\"');  // find first quoted string
    while(quote) {
      if(quote[-1] == '\\') break; // for \" literal quote
      *quote++ = ' ';  // erase quote
      while(*quote != '\"' && *quote != 0)
        *quote++ = ' '; // erase contents of string
      *quote = ' '; // erase ending quote
      quote = strchr(quote, '\"'); // look for next quoted string
    }
    char * cmt = strstr(buf, "//");  // C++-style comments only
    if(cmt) *cmt = 0; // strip comments by terminating string
    puts(buf);  // Look at the modified string
    char * token;  // strtok uses delimiters to find a token:
    if((token = strtok(buf, delimiters)) != NULL){  // first strtok call
      if(strcspn(token, digits))  // ignore constants
        symbol_table.add_or_count(token);
      // subsequent strtok calls for the same input line:
      while((token = strtok(0, delimiters)) != NULL)
        if(strcspn(token, digits))  // ignore constants
          symbol_table.add_or_count(token);
    }
  }  // print the list in 2 ways, using pointers to members:
  symbol_table.apply(&word::print1);
  symbol_table.apply(&word::print2); putchar('\n');
}  // results are output by the destructor calls
#endif // TEST3
```

End Listing

Handling OS/2 Error Codes

Simplify the tracking and correcting of OS/2 development errors with this utility

Nico Mak

This article presents OS2ERR, a simple and effective system to handle unexpected OS/2 error codes in Microsoft C programs. When an OS/2 system function returns an error, OS2ERR displays a pop-up screen with details about the error. It then gives you a choice of aborting or continuing the program that caused the error. The pop-up screen contains the program name, process ID, thread ID, error code, error classification, and the recommended action. It also lists the source code filename, line number, and source code that called the system function. The system error message (if any) for the error code is displayed. This information greatly simplifies tracking and correcting the problem. Figure 1 shows a typical sample output.

This detailed error report saves time during application development and testing, but is generally inappropriate for a final product. OS2ERR was designed to be used in conjunction with typical error handling routines and is easy to "turn off." No source code changes are needed to disable OS2ERR processing — just recompile and relink as described later in this article. Or you can customize OS2ERR for your production environment.

Nico Mak is a software developer for Mansfield Software Group in Storrs, Conn. He can be reached at 70056,241 on CompuServe or as Nico_Mak *on BIX.*

Using OS2ERR

OS2ERR is easy to use. First include OS2ERR.H (see Listing One, page 181) at the beginning of your source program, after including the OS/2 header files. Then use the *os2chk( )* macro on any OS/2 system function to display the pop-up screen if the function generates an error. See Example 1 for an example of how to use this macro. Alternately, you can use the *os2( )* macro on system functions to record the error code, filename, line number, and source line for possible later use in an error handling routine. The *poperr( )* macro will display the information recorded by the *os2( )* macro along with other details about the error. See Example 2 for an example of how to use these macros.

Note that OS2ERR waits until any existing pop-up screen is closed before displaying its own pop-up screen. The *os2chk( )* and *poperr( )* macros should therefore not be used while your application is displaying a pop-up screen.

To enable OS2ERR processing while testing your application, compile your programs with the –DOS2ERR option, and link your application with OS2-ERR.OBJ. If this option is NOT used, then the macros do not generate code,

```
Error code:         6
Program:            D:\DDJ\ERRDEMO.EXE
Command tail:       test
Process ID:         98
Thread ID:          1
Action:             Abort in an orderly manner
Locus:              Unknown
Class:              Application error
Source name:        D:\DDJ\ERRDEMO.C
Source line:        11 (DosRead(hf, Buf, sizeof(Buf), &cbRead))

Incorrect internal file identifier.
Press Esc to abort or any other key to continue
```

Figure 1: *Sample pop-up screen*

```
/* examples of os2chk() macro */
 os2chk(DosClose(hanConfig));    /* display pop-up screen w/error info */
 os2chk(VioGetFont(&viofi, 0)); /* if system calls return error codes */
```

Example 1: *Examples of OS/2chk( ) macro*

and OS2ERR is disabled. When OS2ERR is disabled you do not need to link with OS2ERR.OBJ.

I've tested OS2ERR with Microsoft C, Version 5.1, and IBM OS/2, Version 1.10, and with the Microsoft OS/2 Software Development Kit, Version 1.06. Only one minor change was needed when switching among these environments: The *DosGetPid* and *VioWrtTTy* system functions were renamed to *DosGetPID* and *VioWrtTTY* in the SDK.

You can customize OS2ERR for your production environment

OS2ERR Components and Operation

OS2ERR.H defines the *os2( )*, *os2chk( )*, and *poperr( )* macros. These macros generate calls to routines in OS2ERR.C (see Listing Two, page 181). The *os2( )* and *os2chk( )* macros pass the following information to these routines: the source code of the OS/2 system function, as provided by the ANSI C stringizing operator (#), the file name and line number, as provided by the *__FILE__* and *__LINE__* predefined identifiers, and the return code from the OS/2 system function.

OS2ERR.C contains routines to display the pop-up screen, error messages, and handle the abort or continue prompt. In addition to the information passed by the *os2( )* and *os2chk( )* macros, these routines use the *DosGetEnv* function to access the program name and command tail, the *DosGetPID* function to report on the process and thread IDs, the *DosErrClass* function to determine the system recommended action, locus, and error class, and the *DosGetMessage* function to obtain the system error message text. They use the *VioPopUp* and *VioEndPopUp* functions to manage the pop-up screen, and call *KbdCharIn* to wait for the user to press a key. If the user presses Esc to abort, the *DosExit* function is invoked to terminate the process.

Conclusion

OS2ERR has helped me to quickly identify the source of a number of OS/2 development problems. I hope it also saves you time. If you make enhancements to OS2ERR, I encourage you to upload the modified code to the Dr. Dobb's Forum on CompuServe. I've already uploaded a version that includes the system identifier for the error code (from BSEERR.H) if there is no system message available for a particular error. Comments and suggestions are welcome.

Source Code Availability

As a service to our readers, all source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Drive, Redwood City, CA 94063, or call 800-356-2002 (inside Calif.) or 800-533-4372 (outside Calif.). Specify issue number and disk format. Code is also available through the *DDJ* listing service (415-364-8315) and through the DDJ Forum on CompuServe (type GO DDJ).

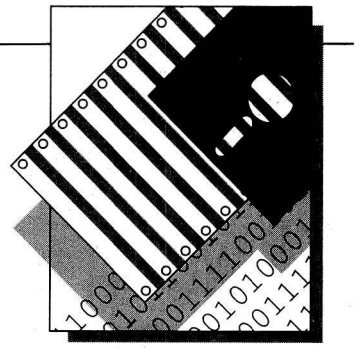
DDJ

(Listings begin on page 181.)

Vote for your favorite feature/article.
Circle Reader Service **No. 13.**

```
    /* examples of os2() and poperr() macros */

SubCommandProcess()
    {

    if (!setjmp(jmpbuf)) {                          /* set up error handler */

        /* ... processing ... */

        err == os2(VioGetFont(&viofi, 0));          /* get current font info */
        if (err == ERROR_VIO_EXTENDED_SG)     /* if running in a VIO window */
            InVioWindow = YES;                 /* remember we're in window */
        else if (err)                        /* if any other VioGetFont error */
            poperr();                /* display pop-up screen with error info */

        /* ... more processing ... */

        if (os2(DosClose(hanConfig)) /* if DosClose returns an error code */
            longjmp(jmpbuf);         /* skip to application's error handler */

        /* ... yet more processing ... */

        }
    else
        poperr(); /* error handler - display pop-up screen with error info */
```

Example 2: *Examples of OS/2( ) and poperr( ) macros*



Windows 3.0 Challenges All the Talent in the Room

Michael Swaine

The title of this month's column is too obscure a reference to stand unexplained.

Thirty years ago in *Advertisements for Myself*, Norman Mailer established his reputation, partly for talent but mostly for arrogance. In a section of the book called "Evaluations — Quick and Expensive Comments of the Talent in the Room," he trashed most of his contemporary writers. Myrick Land documented the ensuing feuds in his less celebrated but funnier book, *The Fine Art of Literary Mayhem*. The Mailer chapter — Mailer must have been miffed to find that the entire book was not devoted to him — was titled, "Mr. Norman Mailer Challenges All the Talent in the Room."

Microsoft spent megabucks rolling out Windows 3.0 this May and will be spending more over the next months to encourage Windows 2 users to upgrade and nonusers to take the plunge. Is this upgrade such a big deal?

It's such a big deal. Bill Gates claims that Windows 3.0 will have an impact on every aspect of the personal computer industry. He's probably right. The following are some thoughts on Windows 3.0 vs. the talent in the room, with the glaring exception of the heavyweight contender out in the hall. As Stephen Morse pointed out in *PC Week*, Presentation Manager is quite capable of delaying itself with no help from Windows 3.0

Win 3 vs. DOS

Industry watcher Stewart Alsop's view is that the release of Windows 3.0 will increase Apple market share along with the Windows market share, both at the cost of DOS. *PC Week* quotes Bill Gates as predicting that Windows will outsell DOS in a year.

Windows as a threat to DOS is, in a sense, nonsense. Nobody is going to buy Windows rather than DOS. Windows is a DOS extension, more closely integrated with DOS than any shell, requiring and using DOS, in particular the DOS file system. Furthermore, Windows 3 supports non-Windows applications, meaning plain DOS applications. Finally, any way you slice the Windows/DOS pie, Microsoft gets fed.

The threat that Windows poses is to DOS as a development platform. It raises the question, is there any reason to develop for plain DOS anymore? The answer is surely yes. Here's a sample of some classes of users who will still need or want raw DOS.

1. The users of dedicated one-application PC-class machines that do their respective jobs well. If such a user sees no need to upgrade an 8086-class machine running DOS and such an application, then the user will probably not make the significant investment required. Some such applications, despite Windows 3's support for non-Windows applications, will need an upgrade, and some perfectly functional applications that happen to be no longer supported won't be getting any upgrade.

It's interesting to speculate on how many of these machines will go on cranking away in such environments. And that's all we can do: speculate. Industry history is not long enough to base predictions on, and pre-personal computer software and hardware experience is an entirely different world of technology and pricing. But the matter is unimportant in terms of developing commercial software. Such machines and people effectively become invisible to the market. One application, already sold, is not a market; people who have no need to spend money are not potential customers. The veil falls on these people.

2. The impecunious. Also behind the veil. To make use of Windows 3.0, in particular to be able to use multitasking, the minimum configuration is a 286 with 2 Mbyte of memory and EGA. Any PC owner who doesn't have this hardware has the choice of buying it or doing without Windows. Some will do without, and like homeless people in the census, will drop out of the demographics.
3. Users of embedded systems. Last fall, Microsoft released ROM Executable MS-DOS v.3.22, a true DOS in ROM. Datalight and Digital Research both also sell ROMable DOS clones. These users probably don't need windows and icons.
4. Engineers and scientists. There should be no compelling reason for anyone in this market to buy or hang on to a raw DOS machine as a CPU; if there is in a year or two, it will mean that Windows developers have not been seeing all the market opportunities. But a raw-DOS machine as a device is another thing altogether. One can easily imagine an array of DOS machines controlling instruments, with a Windows box standing between all of them and the researcher user.

But for new purchases of personal computers for business and other commercial uses, raw DOS is dead. With upgrades running $50 even for runtime users, Windows 3 is priced well, and Microsoft is pushing it hard.

Win 3 vs. other DOS Extenders

The actual competitors to Windows on Intel CPU-based hardware are the other DOS extenders or Windowing systems. Anyone interested in the technical details of how DOS extenders can live with Windows 3.0 should read Ray Duncan's *Extending DOS* (Addison-Wesley, 1990). Theoretically, the DOS Protected-Mode Interface (DPMI) lets a DOS extender attain full compatibility with Windows 3.0. But now that there is a canonical windowing extension to DOS produced and marketed by the DOS folks, it won't be fun to compete in that market. And Windows 3 is much better than Windows of the past.

Win 3 vs. Win 2

Windows 3 is a big improvement over Windows 2. What strikes one immediately is that the Romper Room primary colors have been replaced by a suite of sophisticated color schemes. This is of no technical significance whatever, but it is indicative of the hand of designers in the user interface, and that's very important to the product's success. At the low end, where a big battle will begin this fall, this sort of thing matters a lot.

Microsoft's Windows 3 GUI team included human factors psychologists, and visual designers were involved in the development of the GUI. The result is a product that should be significantly more attractive to users than Windows 2. The user is given full control over color, but it's easier to select a predefined suite of colors that look good together than to tweak individual colors. This is a good example of providing the customizability but guiding the user toward reasonable customizations.

The Windows interface now uses proportional fonts. Not only does this make the display more readable, but it also raises the professional look of the product. Examining installed fonts and installing new ones is simple and intuitive. The next release is expected to use TrueType outline fonts.

Setup and configuration for particular hardware is largely automatic. This is also very important if Microsoft expects Windows to bring new people into computing. Apple, which has an easier job of it, has set up plug-and-play expectations that Windows has to come close to meeting, and does. Also, user-friendly Control Panel options let the user select printer and network options from lists; these selections become effective immediately, without reinstalling or restarting Windows.

There are some holes in the GUI that let DOS show through. The SysEdit program, which brings up all configuration files for editing, is very handy, but looking through a Maclike window at DOS batch commands will be a disorienting thing for a new computer user.

All these user interface issues are important to the user acceptance and the success of the platform. More important from a technical perspective are the questions of what capabilities the product supports, and how easy it is to develop for it.

The big accomplishment is probably memory management. Running in protected mode on the 286 and 386, supporting virtual 8086 machines on the 386, are things that matter. 3.0 is also more cognizant of networks than 2.x, but it doesn't demand a lot of network awareness from the user. New APIs let network applications mask the difference between local and remote resources. Transparency of remote access is going to become important.

It will take time for the benefits of interapplication communication to emerge, but the dynamic data exchange messaging protocol (DDE) provides the basis for applications that cooperate in ways not seen before.

The reassuring news is the ease of transition from Version 2 to Version 3, which, Windows developers are saying, is a piece of cake. The essential tool, of course, is the Windows SDK, and device driver developers will need the DDK. New Windows developers should probably start, according to developers at a recent meeting of the Software Entrepreneurs Forum, with Charles Petzold's Programming Windows.

While the GUI is not as unified and intuitive as the Mac's, it's close enough to make a Windows box look like a direct competitor to a Mac. And technically, Windows has capabilities that go well beyond what the Mac does at present.

Of course, Apple has something up its sleeve.

System 7 vs. System 6

At the Apple Worldwide Developers Conference in May, developers got five days' intense exposure to System 7, the biggest thing in Apple system software since the original Mac operating system. It's expected to be out by the end of the year.

System 7 includes a new Finder and other reworkings of the user interface. The improvements are impressive, delivering more power and customizability to the user while at the same time making the interface more consistent, with fewer things to learn. There's something more than consistency involved;

(continued from page 138)
it has to do with getting across to the user a big idea, a general metaphor, and letting the user figure out how things ought to work. Apple does a better job of this than anybody in the industry, and Finder 7 is better at it than the current Finder.

Now, pretty much anything the user can click on does something reasonable: Applications, utilities, desk accessories, and control panels all launch when clicked; documents open; and other things return useful information about themselves. Some formerly opaque objects now open to reveal their components. This natural extension of the click-to-open idea eliminates the need for one utility and one user interface: The font/DA mover is no longer needed because fonts and DAs (desk accessories) are stored in openable containers.

Windows as a threat to DOS is, in a sense, nonsense. Nobody is going to buy Windows rather than DOS

Not all the changes reduce the set of concepts the user has to acquire. Apple agonized over one element that it has finally added to the interface: The movable modal dialog box. The user interface had not previously distinguished modality from immovability, and separating these aspects visually caused a lot of headaches. The final design, drawing both its visual appearance and performance from existing elements, will probably pass completely unnoticed by most users, which will be evidence that Apple did it right.

There's a Find menu option for finding files; it seems to be fast, and allows a lot of ways to direct the search. There are a number of smaller tweaks to the GUI. Band selection and window and trash behavior have been honed. The new trash can is another example of the tighter integration of the interface: The trash can was formerly a unique object, but now it's a folder, with no peculiar behavior to learn and adapt to. There are new sizes and colors of icons. The outline views in folders and the new keyboard equivalents for mouse moves look like similar features in Windows.

There's also a system-wide help system called "Balloon Help." Turn it on via an ever present menu item, and anything you point at gives some helpful account of itself in a pop-up cartoon-style word balloon. Apple has already documented the system itself with Balloon Help. Developers can add Balloon Help to their applications without changing any code; the help messages are just resources with pointers to the resources (menu items, controls, window parts, and so on) that they document. Balloon Help is one aspect of the Toolbox Help Manager, which provides facilities for more customized help systems. Balloon Help is intended to provide a basic level of help, letting the user ask, "What is this?" or "What does this do?" It's not intended to be the basis of a complete on-line documentation system, but Help Manager is intended to provide the tools for building such a system.

System 7 includes virtual memory, which the user can turn on or off. MultiFinder is always on, so the System 7 Mac is never a one-application machine. Finder, the system-support application, is always on, reachable via an everpresent menu at the right end of the menubar, its capabilities available to other applications via Apple's interapplication communication.

The interapplication communication (IAC) and networking features are nice. IAC includes several components, particularly AppleEvents, Edition Manager, and the PPC Toolbox.

AppleEvents and changes to the current Event Manager implement the framework for a standard language of messages that applications can pass to each other or to the Finder. The set of standard events is largely undefined; Apple is soliciting developer input. A few events have been defined, including Oapp (open application), ODoc (open document), PDoc (print document), and Quit. All future applications are expected to support these, and others that are being defined. A set of events called "Finder Events" has been defined and implemented in the Finder; applications can avail themselves of the capabilities of the Finder by generating Finder Events.

The Edition Manager implements the publish and subscribe capability. This is live copy and paste: When the "published" document or document section is updated, all subscribing documents are also updated. Apple wants all applications to support publish and subscribe, making the facility as ubiquitous as copy and paste. There currently are some user interface issues to be resolved; even Apple's own HyperCard

(continued from page 140)
team appears to be holding off on support for Edition Manager until these issues are resolved.

The PPC Toolbox is the low-level facility for implementing process-to-process communication. Since there is no standard set of events at this level, it will require close cooperation between application developers to use this level. Because it's low level, it will provide higher performance than AppleEvents. It will probably be used by large software vendors to put high-performance integration into suites of applications. This superiority of performance and integration will probably be used to sell the idea that users ought to buy all their applications from one vendor. Only the big guys will be able to make this pitch.

Win 3 vs. System 7

It seems likely that the things that Windows 3.0 and Mac System Software Version 7 have in common will define personal computer application development in the '90s.

Graphical user interfaces are here to stay. A serious push in the low end of the market by clone makers bundling Windows and Apple trying to redefine itself as a company will make it very hard to sell a DOS box. I've hinted at this low-end war, but anyone who reads the weeklies or heard John Sculley speak at the Developer's Conference this spring knows that Apple is claiming to be ready to accept significantly lower margins to compete at the low end with a line of high-volume, low-cost Macs. These machines will all run System 7 and will not, unless Apple blows it badly, be feeble machines. They will be competitive in price and performance with 386 clones running Windows 3.0.

Multitasking, interapplication communication, and transparent network accesses will result in new kinds of applications. Large vendors will develop interlocking suites of applications rather than integrated packages. Smaller vendors will develop small, targeted applications. Utility developers will come up with enhancement products that ride on other products, sending messages to these products to get their work done. Although this last scenario sounds a little like spreadsheet macros, there is a difference in control. A spreadsheet macro runs within the spreadsheet application and is vendor-specific; these utility applications use other applications but are separate, clickable applications, and they should work with any applications that support the AppleEvents they generate.

Overall, Apple and Microsoft are saying, we'll see more, smaller applications.

Applications will, more and more, take on a tool aspect. What, exactly, does this mean? Like all the preceding predictions, this is just what Apple and Microsoft are telling developers. It would be an uncharitable exaggeration to say that the message Apple and Microsoft have for third-party developers is: "We'll talk to the users; you just talk to us." That's not what they're saying. Nevertheless, if the model of interprocess or

Applications will, more and more, take on a tool aspect. What, exactly, does this mean?

interapplication communication that Apple and Microsoft are pushing comes fully into existence, it would be conceivable for a user to use a product without ever seeing its user interface.

Nobody is talking about what "more smaller applications" implies in the battle for computer store shelf space.

The differences between Windows 3 and System 7 are less important than the similarities. Apple has the better user interface, but it's not clear that that will make any difference to anyone but current Mac users. I can't see a Mac user converting happily to Windows, but the superiority of the Mac interface may not be great enough to affect buying decisions by present PC users or new computer buyers.

One significant difference between the two systems has to do with their file systems. Windows 3.0 still uses the short DOS file names and sometimes requires the user to deal with DOS path names. More objectlike file management tools are expected in the future, but right now the file handling has some kludgy aspects, and the DOS file names are unfriendly.

Mac's System 7 has a new file tracking system called "Alias Manager." On the surface, Alias Manager lets users create aliases, which are small files that refer to other files. Create several aliases for an application and you can store the aliases in different folders. Click on any of the aliases and the original application gets launched. The user doesn't have to worry about paths, because the Alias Manager keeps track of the original even if it or the aliases are moved or renamed.

The Alias Manager does more than permit aliasing of files, though. It is, as of System 7, the tool of choice for keeping track of files for any purpose. The standard file facility will use it, so any application using Standard File will get the benefits of Alias Manager from it. Alias Manager can be used for a wide variety of purposes. For one example, you put an alias for every data file on your hard disk into a folder called archive, then archive all the data files to floppy disks. Now when you click on the alias in the archive folder, the Alias Manager will prompt you to insert the appropriate diskette. Since the alias files themselves are very small, this lets you catalog all your diskettes. Aliasing volumes lets you kludge a simple network manager. The highlight of the Alias Manager session at the Developer's Conference was the simple Adventure game created on the desktop without any programming simply by aliasing files and folders.

The alias manager works by maintaining alias records in memory, and employs a whole battery of heuristics for identifying the target volume and the target. It is not easily confused by mere renaming or moving of the target, and will search across network zones and request mounting of volumes if necessary.

The discussion of Mac OS and Windows similarities and differences ought to lead to a discussion of porting strategies, or to a discussion of techniques for developing one program for both environments. A couple of years ago, Michael Brian Bentley wrote an interesting but intimidating book called *The Viewport Technician* (Scott, Foresman, 1988), about writing software for multiple windowing platforms. If it's getting easier to do that, and to end up with efficient programs, that would be news, but it's not clear that it is.

ToolBook vs. HyperCard

The porting problem has been addressed for two applications that will be bundled with Windows 3 and System 7: ToolBook and HyperCard. ToolBook, from Microsoft co-founder Paul Allen's company Asymmetrix, is a HyperCard-like software construction kit that requires Windows 3.0. ToolBook substitutes the page-in-book metaphor for HyperCard's card-in-stack metaphor and has some more concrete differences with HyperCard as well, but it is definitely intended to be the HyperCard equivalent for Windows. The latest version of HyperCard, 2.0, has some powerful features, but it still lacks some of the capabilities of ToolBook. The products are enough alike that Heizer

Software has already announced a product that converts HyperCard stacks into ToolBook books. I haven't seen it yet.

I have seen ToolBook, and I've seen HyperCard 2.0. I reviewed ToolBook in *Personal Workstation* magazine recently, and will be writing more about it and HyperCard 2.0 here soon. What follows here is a look at how "the hypertext problem" is handled by ToolBook and HyperCard 2.0.

In my June column I talked about ways in which some people have used HyperCard and HyperTalk to do research into what could be called the hypertext problem. The problem is how to implement links in text, and there is as yet no consensus on the proper way to do it. After writing that column, I got my hands on HyperCard 2.0. Apple did a better job of leaving the developer's options open than I expected when I wrote that column, but still didn't provide everything the hypertext author might want. Meanwhile, Windows 3.0 came out, and Asymmetrix finally got to release its ToolBook. ToolBook runs under Windows 3.0 and is very comparable to HyperCard 2.0, but its approach to the hypertext problem differs fundamentally from HyperCard's.

The basic problem is this: Given a body of text, we want to be able to permit the reader to click on part of the text and have something happen. The "something" is usually the display of further information on the subject covered in the clicked-on text. This is information that doesn't fit into the linear structure of the running text: a definition, an optional detail, a digression. The further information may be displayed in any of a variety of ways: As a replacement text that takes the place of the clicked-on text, in a pop-up field overlayed on the clicked-on field, or in another body of text like the original body.

What I call the hypertext problem has two parts. The first part of the problem is how to indicate to the user what can be clicked on. In linear, printed text, there are established conventions (for example, parentheses and footnotes) for signalling a departure from strict linear structure. The footnote convention is the closest to what goes on in hypertext systems: We use a superscripted character to signal the existence of a link to a block of text at the foot of the current page. Among the techniques proposed for signalling hypertext links are superscripted characters, font changes, and boxes around text.

The other part of the hypertext problem involves that imprecise phrase, "permit the reader to click on parts of the text." What could a hypertext author mean by "part"? Must every link be tied to a single word? Footnotes in linear printed text often refer to entire sentences or paragraphs, although there is often no visual cue to suggest this. Can the clickable zones in a hypertext document overlap; that is, can you arrange that a sentence and a word in the sentence are each independently clickable, each invoking a different text link? Must clickable text be contiguous?

But this second part of the problem gets a little deeper than this. Does the link pertain to a particular instance of the text, to any appearance of that text in the text field, or to the nth word in the field, whatever it might be? For particular hypertext purposes, any of these might be desirable.

Case 1: Linking off a particular instance of a particular string. If you want the link object (the clickable text) to have substance, so that it remains a functioning link even when you move it somewhere else, you probably want something like this.

Case 2: Linking off any instance of a particular string. One excellent use of hypertext links is to provide definitions for technical terms in a document. The author does not want to force the reader to look at the definition the first time a particular term is used, but wants to allow the user to look up the word by clicking on any instance of the word in the document. The logic is that, for any particular use of the term, the user may be able to work out part of its meaning from context or outside knowledge, but may still need a definition later when the term is used in a different context. For this or a variety of other reasons, a full hypertext authoring system ought to allow the author to create a single link for all instances of a particular word or phrase.

Case 3: Linking off whatever string occupies a particular position. "Position" here refers to text position in a document, not pixel location on a screen. This approach allows the author to create links that depend on the structural features of a document. While this might not be useful for casual text, it might make sense for highly formalized documents, such as a block of assembly language code.

This is one way of slicing up some of the hypertext territory; these cases don't cover all the possibilities or represent the only way to segment the possibilities. I present them only to give some framework for looking at what ToolBook and HyperCard 2.0 actually do.

ToolBook and the Hypertext Problem

ToolBook has two main ways to show the clickable text to the user. First, the cursor changes when passing over this text. Second, there is a facility for making all the links visible. One convenient time to do this is when the cursor passes over the text field. This technique puts a box around each instance of linking text. These two techniques are well thought out. The cursor change does not use any element that might have another meaning in the text, such as italics, and is perfectly obvious if you're looking for it without being terribly intrusive if you're not. The boxing also does not use a feature that you would want to use for other text purposes. The boxes could clash with other framing devices, and could look ugly, but since it can be toggled on or off easily, this is not too serious. It's chiefly a peeking device for the user. You could also use font-style change to signal a link, since ToolBook lets you mix fonts and styles within a field, something that HyperCard formerly did not do.

ToolBook implements links via hotwords. A hotword is an object with properties, a script, and a place in the object hierarchy of ToolBook. You create a hotword by selecting text and choosing a menu option that makes that text a hotword. A hotword is a Case 1 link, a link off a specific instance of text. You can move or copy it and the link moves with it, because the link is defined by the script of the object. You can also edit the text of a hotword, although if you delete all its text, the object disappears.

HyperCard 2.0 and the Hypertext Problem

In June, I documented some of the contortions that HyperCard stack developers went through to kludge some sort of hypertext facility in HyperCard. Because HyperCard and ToolBook both have programming languages, that sort of roll-your-own hypertext is always possible. There's probably no approach to hypertext that can't be implemented, some way or other, with each of these products. But hypertext systems have to have reasonable performance, and that means that the hypertext abilities of these products depend on the techniques built in. With Version 2.0, HyperCard has more hypertext capabilities built in than it formerly had.

In the area of showing the clickable text to the user, HyperCard puts all the decisions in the hands of the developer. Since there are different purposes for hypertext and the jury is still out on the user-interface decisions, this is generally the smart approach when you want to provide a flexible hypertext authoring system. But the flexibility Ap-

ple offers is chiefly in font and style change, which is a usurpation of the features of text for the purposes of hypertext. One option for marking links that would not be at all easy to implement in HyperCard is the ToolBook box, and messing with the cursor in HyperTalk could be problematic.

HyperCard implements links via field handlers. Text can be grouped, providing some of the capability of ToolBook's hotwords, and a field script can examine attributes of the clicked word, clicked line, or arbitrary grouping of clicked-on text. It's possible to test font or style and link off it, which is an interesting reversal: Instead of highlighting the link, you link off the highlight. HyperCard's field handler approach to hypertext means that you can key off any instance of a particular word or text with a single handler (Case 2). Case 3 links are also fairly directly supported by this scheme, but not Case 1 links.

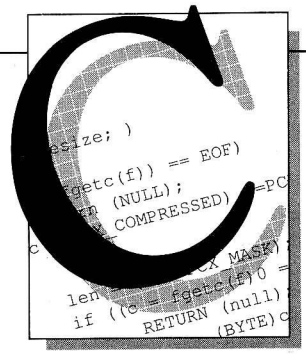
Here are my closing thoughts on ToolBook, HyperCard, and the hypertext problem:

The indication of links: This is hypertext; i.e., more than text. To indicate the links, the author must add something to the text, and in particular to the visual appearance of the text. I'm not sure that it is widely understood that the indication of links raises spatial design problems. Writers have not in the past had to deal with such issues, but hypertext authors, because they are working in more than one dimension, do have to deal with spatial design in their documents. It is up to the hypertext author to create and indicate links without making the document ugly and unreadable.

What to link from: Between the two of them, HyperCard and ToolBook cover the linking techniques I somewhat arbitrarily defined here. In my opinion, both of them ought to provide the tools for creating any of the types of links I've described, or any reasonable kind of link anybody might come up with, if they are to serve as general hypertext authoring systems. Neither, of course, is being marketed as a general hypertext authoring system, but many people will be building hypertext documents with them. And until someone demonstrates an unequivocally "best" way to implement hypertext links, the hypertext author should not be constrained to one or a few techniques.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 12.**



The Past, the Future, and Multi-mania

Al Stevens

The annual C issue is a special one to me. Besides being devoted to C, the issue is also my anniversary with *DDJ* as the C columnist. August 1990 starts my third year, and marks the time for a retrospective view of the past two years. A lot has happened. In that time we watched C strengthen its position as the preeminent software development language. The major milestone in the advancement of C was the approval by ANSI of the standard definition early this year. It was a long time coming. Second place goes to the gradual swell and then sudden explosion of interest in and use of C++.

Other milestones. The Integrated Development Environment established itself as the preferred way to write and test C code. Source-level debuggers got bigger and better. Developers of operating environments began producing Applications Program Interface libraries so that C programmers could write code into those environments more easily.

None of these developments actually started in the last two years. They have all been around for a while. But each of them really took hold in the very recent past.

Several important new compiler versions came out in the first half of 1990. Borland announced Turbo C++ in May. Microsoft released Version 6.0 of the Microsoft C compiler. Watcom released Version 7.0 with 8.0 coming soon. Zortech released their C++ Version 2.0. It is important to consider as well what did not happen in that period. Neither Unix nor OS/2 took over the PC marketplace. Graphics user interfaces did not replace text-based applications. Every workstation does not have a CD-ROM. Programmers have not been replaced by application generators or so-called fourth-generation languages.

The Future

You do not need tarot cards to guess what C programmers are going to be doing in the next several years. A prevalent platform target for programmers will be, I believe, workstations operating in a network. For the foreseeable future, at least, the workstations will be MS-DOS machines, and the networks will be NetWare. There is nothing bold about that prediction except that it might raise the dander of those who believe or who would prefer that it is inaccurate.

The position of the DOS machine is secure. The acceptance of the NetWare environment is growing. They are a natural fit because a NetWare system can readily accommodate software that runs as well on a single-user PC. Log into the network, and your DOS application takes on the properties of a multiuser system. Usually. Many of us will be writing DOS applications that can sense the presence of a network and deal appropriately with it. To prepare we need to understand the environment and look at some of the tools that support it.

Multiuser

Multiuser systems in the past had multiple terminals connected to a single computer with a multiuser operating system. Unix is such a system. Each user had a terminal. One processor executed the programs for each user. The processor jumped from user to user by giving slices of CPU time to each terminal.

Today the terminals are computers themselves. They run their own programs. A network architecture takes advantage of that fact by off-loading to the workstations the work that is done specifically for the user and letting the host machine, the "file server," perform the common and shared tasks. The multiuser aspects of the system are managed by the file server.

A network file server maintains and stores the system's shared resources, including common data files, software, and printers. The file server can also perform services that are common to all users. One example of such common services is electronic mail. Another is printer sharing.

An advantage to the network architecture is that when users disconnect from the network or the server goes down, the users are still in business because their workstations are standalone computers. If you write a network-cognizant program that can run in the network or by itself, then you have made good use of that advantage. Besides supporting network users, your program can be run by other users who do not have networks.

Multitasking

Multitasking is when a user can run more than one program from the same workstation. In the old days a user started a program from the command line at the terminal. The operating system in the distant computer started it up. If the

program was interactive (communicated with the user via the console), the user could call up the operating system command line with a special hotkey. If the program ran in the background, the OS returned the terminal to the command-line prompt as soon as the program was underway. In either case the user could start a second program and the two ran together in tandem.

Some personal computers have multitasking operating systems. The Amiga has AmigaDOS. The Tandy Color Computer uses OS-9. The PC's DOS is not a multitasker, which is why TSRs were invented. But a DOS user has a choice. If the PC has enough extended or expanded memory, a DOS user can add a multitasking shell to DOS and have all the benefits of the multitasking environment. DesqView is one such shell. I am writing this column with XyWrite and DesqView 386. At the same time I am compiling a large C system and exchanging electronic mail messages with kindred souls on CompuServe. Do not let anyone tell you that you cannot multitask a DOS PC.

If you write programs to run in a multitasking environment, you can take advantage of that environment by writing parallel processes that are aware of one another. For example, a database program can spawn a search process as a separate task and return to the query composition program. The user can be writing another query while the first one is being processed. You can use the facilities of the multitasker to synchronize processes.

The main purpose for the file server is to permit users to share the server's disk space and to share files

Mix and Match

Assume that you target your next application for workstations with DesqView running on a NetWare network. You can ignore those environments and write a DOS application and it will probably run OK. It will, that is, unless you completely ignore the fact that more than one user could be running that program.

There are things to concern you when you run a program on a network. Suppose the program has one configuration file with a fixed name in a fixed place. All your users are stuck with using the same configuration of options because each of them cannot have his or her own copy of the file. A network-aware program does not build in such restrictions.

There are things to consider about a program that runs in a multitasking environment. The closer the program gets to the hardware, the less likely it is to peacefully coexist with other programs. Do not confuse TSRs with multitasking. A TSR takes over the whole machine by tricking DOS into thinking the TSR is the only program running. It changes the environment to suit itself and puts things back the way it found them when it is done. A multitasked program runs in a virtual DOS machine within the context of the multitasking shell. Other programs run in that context concurrently. If your program reprograms the keyboard, reads and writes the mouse ports directly, and writes into video RAM indiscriminately, it can mangle the environment for the other

(continued on page 153)

(continued from page 150)
programs running in the same physical machine.

Writing an application that is well-behaved in these environments and that will work as well in a vanilla DOS PC is no easy feat. You'd be nuts to tackle it without help. Quarterdeck Office Systems, the vendor of DesqView, and Novell, the vendor of NetWare, both provide C programmer's API packages to allow you to integrate the functions of those environments with those of your C programs. This month we discuss the NetWare C Interface-DOS package.

The NetWare Environment

A NetWare network consists of a file server and a number of DOS workstations. The file server is usually a PC dedicated to its server functions, and it runs unattended. The workstations each have a network card that is cabled to the file server. There are different electronic standards for the connection and different cabling conventions.

The main purpose for the file server is to permit users to share the server's disk space and to share files. A secondary purpose allows users to share printers. You can, therefore, build a network where most of the disk space and all of the printers are on the server. The workstations have the minimum hardware needed to run the programs. All the software and most of the data files are on the server.

Once the server is running, each workstation runs a TSR program called the "network shell." Its purpose is to communicate between the server and the workstation. The shell observes how many disk drives the workstation has and maps higher-drive letters to the volumes on the file server. Then it intercepts DOS calls. If a program is opening, closing, reading, or writing a file on one of the server drives, the shell exchanges packets with the server. Otherwise it passes the calls through to DOS on the workstation. The program usually does not need to know the difference. Sounds simple enough. In fact, the environment is much more complex than that.

Ideally, a user would be unaware that the network is operating. Except for some additional commands, the user interface is the same as DOS. But there are some internal things that can muddy up the process and that a programmer needs to know about. A high-level user called the "network administrator" assigns passwords and grants privileges to users to restrict access to server file subdirectories. The shell can intercept printer output and redirect it to a network queue for spooling.

Users log on with "userids." Your user can "shell out" to DOS from inside your application and log off of the network, perhaps logging on under another userid with different access privileges. Files that were available a moment ago, now seem — to your program — to not exist. Programs need to be aware that such things can happen.

The NetWare C Interface

If you are going to write a program that runs in a network, you need to know how to make the extended DOS calls that manipulate the network itself. The NetWare C Interface is a library of C functions that provide that interface. I am not going to try to discuss the entire API here. That would be a big book all by itself. Instead, I will address a few small areas to give you a taste of how the platform works and how well the API is implemented and documented.

Programs that run in a NetWare network will be one of three types. First is the program that knows nothing about NetWare. Programs in this category are DOS programs written for the single-user PC. Most of them run OK on the network. Sometimes you will find a program that opens a file in a fixed place — usually the subdirectory where the EXE exists — and keeps it open. A

second user on the network cannot run the program until the first user's copy terminates, because NetWare will not allow two users to have the same file open at the same time. Some installations simply run such programs from the workstation's local disk and bypass the network altogether. Others use utility TSR programs such as Net-Aware to intercept DOS open calls and substitute user-defined paths and names.

Files that were available a moment ago, now seem — to your program — to not exist

The second category of program is one that will run in a network or as a DOS program in a stand-alone PC. WordPerfect is an example. The program takes advantage of the facilities of the network when they are available and gets along without them when they are not. Printer selection is one such facility. If the program senses that no network is running, the program simply uses the facilities of DOS.

The third category of program is the one that runs only in a NetWare network. It uses network facilities that DOS does not support. A user-to-user message and chat system would be such a program.

You don't need any help writing the first category of program. But to write one of the others, you need to use the NetWare API. Before the NetWare C Interface was available, the API consisted of a document that explained the API calls at the assembly language level. NetWare API calls consist of an extended set of INT 0x21 functions that the shell intercepts and processes. Most of my NetWare programs in the past used C language interrupt calls to invoke the API functions.

You can still take that approach, but it is one intensive pain in the nether quarters. Every call has its own unique format for request and response packets and you will code a zillion different structures to deal with them. If you get the slightest element wrong, either in format or content, the API either returns a meaningless error code or freezes the machine. You do not want to work that way. The C Interface functions take care of the details by hiding the packet formats and providing parameter lists and return values to invoke each API function.

There are, of course, some problems. The API documentation is sprinkled with errors of commission and omission, ones that will trip you up from time to time. A CompuServe subscription and membership in the NOVA forum is a must. There are always knowledgeable folks there who can answer your questions.

Using the API requires that you understand the internal architecture of the NetWare operating environment. The API libraries are divided into 17 categories ranging from Accounting Services to Workstation Services. There is a data base called the "Bindery." There are transaction tracking functions, a queue management subsystem, and soon, a print server API. To figure out what all these categories are you need to read the "System Interface Technical Overview" document and then experiment some. It is not always clear from the descriptions of the functions what their purposes really are. Further, if you want to do something that does not have a corresponding API function, you need to piece together the logic for the series of functions that will support your requirement. There is not a lot of help in the API documentation for these kinds of analyses.

Suppose, for example, that your program wants to display the userid of the user who is logged onto the work station. (I had just such a requirement for an e-mail program.) A search through the various API services reveals no apparent function that returns that piece of information. Keep digging. What you eventually find is a function among the Connection Services called *GetConnectionInformation.* One of the data elements you pass it is a pointer to a null-terminated string to receive the "name of the Bindery object logged in at the connection number." Bindery object? Connection number? A careful reading of the Bindery Services documentation reveals that the Bindery is a database of objects, one of which can be a User. What's the connection number? That's the first argument to the *GetConnectionInformation* function. Where does it come from? Nothing says, so we go searching. Nearby we find the *GetConnectionNumber* function, which returns the connection number that the file server assigns to a workstation. From this bit of secondary deduction we conclude that a function to return the current userid would look like Listing One (page 168), getuser.c.

As an experiment, I wrote the same

(continued on page 157)

(continued from page 154)
program by using the assembly language API and the *intdosx* function that has become a de facto standard among PC C compilers. Listing Two, page 168, is getuser1.c, and it offers an interesting contrast to the C API. Even this small example shows how much more readable the C version is. On the other hand, the version of the program compiled with the C API takes almost 4K more than the one with its own interface. Everything costs something.

Are these excursions through the oblique world of the NetWare architecture typical? In my experience, they are. For example, the documentation for the *GetConnectionInformation* function says that the shell uses this function to see if it is already loaded, and that other programs should, too. But the function needs a connection number, and no connection number exists without the shell already being loaded. Apparently you use any old connection number between 1 and 100, and if the function does not put something in the arguments, the caller can assume that the shell is not there. The documentation does not spell that out, however.

The Bindery

NetWare programmers should be familiar with the Bindery, a general-purpose database that NetWare uses and for which it provides support for applications use. The Bindery is the home for definitions of users, user groups, queues, servers, gateways, and so on. A Bindery entry is an "object." Each object has an object "type" and can contain one or more "properties." A property can be an "item," which can contain one "value" or it can be a "set," which contains a list of values. Apparently Novell is rewriting the database lexicon.

NetWare defines certain object types for the things it keeps in the Bindery. Application programs can add their own object types. As an experiment, let's look at the OT_USER Bindery object type. You can scan the Bindery for objects of a certain type, all objects, objects that match a specified name, and objects with a given object identification number. Listing Three, page 168, is showusrs.c, a program that scans the Bindery for all registered users and displays their userids on the console. To scan the Bindery, you pass an initial objectid of -1 and a name with a wild card (*) to the *ScanBinderyObject* function. The function returns the object's name and objectid. Subsequent calls to the same function accept the objectid filled in by the previous call. This scan continues as long as the function returns the SUCCESSFUL return code. The documentation is vague about what stops the scan, but I figured it out by experimenting.

Incidentally, the SUCCESSFUL return code is zero, the same value that the function returns if the shell is not loaded.

The C API includes functions that account for the use of different services in the network

Try the program without NetWare, and you go into a loop displaying garbage. So, you see, you do need a way to tell if the shell is loaded and the network is operating. I fooled around for a while with the *GetConnectionInformation* function attempting to find a reliable way for it to tell a program that the shell was or was not loaded. No luck. Some time ago I found a different technique, which you can see in Listing Four, page 170, nwloaded.c. It calls the shell at the lower level by using the System Calls API to set the NetWare lock level. The effect and meaning of the calls are unclear, particularly when viewed in the light of what those API functions are supposed to do, but the function works with no apparent ill effects.

Queues

NetWare includes a queue management system (QMS). Queues are lists that the queue manager maintains and they exist as Bindery objects. Users can be defined as being queue users, queue operators, and queue servers. A queue user can add an entry to a queue and observe the status of queues. A queue operator can modify queue entries. A queue server can retrieve entries from queues and service them.

A queue entry is a job. The job is whatever you want it to be. The queue server will perform the job based on the presence of the entry in the queue.

NetWare uses QMS to manage spooling to network printers. The API provides a number of functions that allow an application to control how that printing will occur. A future API will permit you to write print server applications so that you can print from NetWare print queues at workstations. By study-

ing the API interface between print queues and QMS, you can see how you might use QMS in your own applications.

VAPs

A Value-Added Process is a program that runs in the file server. The VAP is for use on NetWare 286. This is a weighty subject, far beyond the scope of this column, and it makes the brave quiver because VAPs are difficult to write and downright thorny to test. NetWare 386 has an improvement on the VAP called the "NLM." The C API provides functions that you use in VAP development, but does not mention the NLM.

A VAP executes when the file server starts up. You have to bring the server down to reload a new VAP. VAPs run in 286 protected mode, which means that you cannot test them with the usual debugger techniques.

Other API Services

The C API includes functions that account for the use of different services in the network. You can write an accounting package that will distribute costs across users. There are API functions to support exchange of data files between DOS and Apple workstations. There are communications and message services for inter-user and inter-network data packets. There are connection, workstation, file and directory services, and on and on.

Summary

I have barely touched the surface of the NetWare programming environment. Of course, it is not possible to cover all bases in the space of a column. The purpose of this coverage is to expose you to the environment, show you the tools, and suggest that this might be the coming thing.

On the whole, I prefer using the C API to the alternative method of coding the low-level API calls into my C programs. The problems I addressed in this column are not problems with the API software itself, but with the level of information imparted by the documentation. Like any other complex software development platform, the NetWare environment takes a while to learn. Programmers who have experience with these functions would intuitively know the answers to the questions I pose. But it takes a while to reach that level of knowledge.

Probably my severest criticism is of the packaging. Novell uses double slip covers with two ring binders in a cover. Even my slender pianist's fingers are not deft enough to easily pull a manual out of its box. Usually when I finally get it out, the rings have popped open, and the pages fall on the floor. Yet the binders are that lay-flat kind with the rings offset on one side, and they cannot be stored outside the slip covers. Doesn't anyone ever try these things out for a while before they commit a big budget to using them?

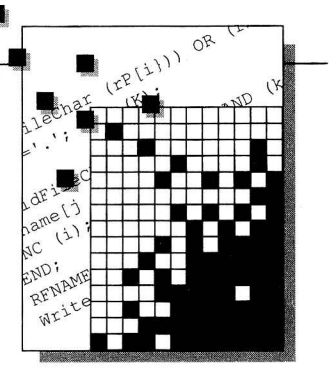
When the demand for better NetWare tools reaches a level that justifies an investment in their development, software houses who cater to programmers will respond. Then you will see third-party function libraries that provide a higher level of access to the NetWare innards, providing an easier interface for programmers. Perhaps someday there will be C++ class libraries with classes for objects in the Bindery, queues, the communications packets, and the rest.

In the meantime, if you will be writing NetWare-savvy programs, you will want to get the NetWare C Interface-DOS libraries from Novell. If you want to study or use the underlying assembly language calls, get the NetWare System Calls-DOS package.

DDJ

(Listings begin on page 168.)

Vote for your favorite feature/article.
Circle Reader Service **No. 13.**



You See Toothpaste Pumps; I See Coil Forms . . .

Jeff Duntemann, K16RA/7

Lord knows, there shouldn't be places named "Drug Emporium" (what sort of message does that give our kids?) but I confess I was in there the other day, looking for some Poore's Potato Chips and the latest *Computer Shopper*. It's definitely the category-killer store of its kind in Scottsdale, with whole aisles devoted to things that might get five shelf-feet in Safeway. I trucked down the toothpaste aisle, oblivious until I hit the toothpaste pump section. Something stopped me cold — bear with me on this; I'll explain — in that all those toothpaste pumps sitting side-by-side on the shelves looked just like plug-in shortwave coil forms.

Sure. Hey, if you're over forty you might remember: Shortwave radios used to use these little things with plugs on the bottom and wire on them, that plugged into a hole in the back/front/side/whatever of the shortwave radio. Ham radio operators used them until fairly recently, and some of us incorrigible atavists use them until this very day. They always came in groups of four or five, one for each shortwave band, hence the feeling of deja vu in seeing 17 pumps of Ultra Brite side by side, standing on end as we always stood the coils on end, flared side down, ready to grab if we got tired of the BBC and wanted to corrupt our brains with Radio Moscow.

I haven't built many radios lately (staying alive has been challenge enough) but I bought a pump of Ultra Brite anyway, and in another six weeks, once the stuff inside is gone, I'm gonna ram an old octal tube base up its business end, wrap some #22 enameled wire around it, build a 1-FET regen and go sniffing for the "Voice of America."

It's a talent I have. I see extraordinary uses for ordinary things. You see scrap plywood; I see a potential birdhouse. You see toothpaste pumps; I see coil forms. I have a garage full of ordinary things awaiting their extraordinary applications. (Some have waited for many years. Ask Carol about it; she'll have plenty to say. . . .)

Bingo!

I've used this talent often in programming, where the difference between the ordinary and the extraordinary is lots fuzzier, and the material itself almost infinitely mutable. I guess you could say that everything in programming is a potential birdhouse; if you need a birdhouse badly enough you'll figure out a way.

Last week I was facing a data entry problem I hadn't faced before. I had to design a means to enter and store magazine bingo cards. You know what I mean; those Free Information! cards in the backs of virtually all magazines (including *DDJ*) where you circle an advertiser's number and they magically send you their full-color brochure. Magazines get hundreds or (God help us) thousands of these in the mail all the time. Each card must be entered into a database somehow, starting with the reader name and address but ending by somehow encoding every number the reader has seen fit to circle.

How to encode a circled number was an interesting question. I started out by taking the low road and creating a 255-character text field in Paradox, into which each circled number would be typed, separated by spaces. Paradox has good pattern-matching abilities in its query form, and it worked tolerably well once you swallowed the data entry. That is, it worked — until some joker mailed us a card with the whole block of 200 numbers circled.

That drove home the fact that 255 characters would not contain much more than 60 or so numbers. Half a second's thought should have told me that the bingo numbers weren't numbers anyway. They were Booleans, either True (circled) or False (not circled.) And our bingo card was just an array of 200 Booleans. The birdhouse I had to build would be an array of 200 Booleans with some means of screen display and editing. I took a look to see what I had lying around in the line of potential birdhouses, and it hit me: A Pascal set is (with a little reworking) an array of 256 Boolean values. Packed, too — in 32 bytes, with not a bit of wasted space. Bingo!

Get set

Apart from *SET OF Char* (which I use incessantly for text filtering) I haven't done much with sets of more than five or six elements. But consider:

```
TYPE
  BitSet = SET OF 0..255;
```

This definition gives us a bitmapped data item containing 256 flag bits, with machinery built right into Pascal for adding, removing, and testing for presence and absence of individual bits. These two statements amount to pretty much the same thing:

```
MyBooleanArray[17] := True;
MyBitSet := MyBitSet + 17;
```

The + symbol here is acting as set union, not addition. (We have overloaded operators in Pascal too! Kind of like discovering that you've been speaking in prose all your life. . . .) In the second statement, you're assigning

(continued from page 157)
to *MyBitSet* the union of the earlier state of *MyBitSet* and the value 17. Think of 17 as an enumeration constant here, and it should make sense.

Similarly, you can remove an item from a set by using the set difference operator, - , like this:

```
MyBitSet := MyBitSet - 17;
```

Again, this may look odd unless you remind yourself that 17 is not acting arithmetically here. Call 17 by its Martian name, Foobity-Foo (see my book, *Assembly Language From Square One* before calling out the guys with the butterfly nets) if you have trouble divorcing it from the realm of practical math.

The bottom line: As the central data item in my model of the bingo card, I would use a Pascal set of the cardinal numbers from 0 to 255. Each number would be either in the set (circled) or not in the set (left alone.)

Breaking Your Own Rules

As is my wont these days, I immediately forged an object around the *BitSet* data item. In doing so I broke one of my own personal rules that I shared with you in a previous column: Don't make an object out of anything special-cased by the compiler. This would include most primitive data types including Booleans, characters, strings, and — most emphatically — sets.

The advice I gave before was and is still sound. Nonetheless, I consider the exception to be good practice because I'm not really using the set as sets are generally used. In effect, I'm using a set as a packed array of Booleans, so just this once, I'll build an object around a Pascal primitive data type and not feel guilty about it.

At the highest level, designing an object consists of asking the question: What must the object know how to do? For my application, it wasn't a long list. The object must be able to clear itself (that is, set all bits to 0) display itself, and edit itself. That's all.

The resulting object is implemented in Listing One (page 172), SETOBJ.PAS.

A Set Object

The visual expression of the set data consists of 16 lines of sixteen three-digit numbers, starting with 000 and going to 255. When a numbered bit in the set is set (for example, binary 1) the three digits are displayed in reverse video. For all bits that are cleared (binary 0) the three-digit numbers are displayed in normal white-on-black video.

Within the *SetObject.Edit* method, a text cursor is created by poking the two triangle symbols (ASCII 16 and 17) to the screen on either side of one of the 256 numbers indicating bits in the set. I call the bit so indicated the "hot bit." When you press Enter, the state of the hot bit is toggled from cleared (the default) to set and back. By bouncing the cursor around the matrix using the arrow keys and pressing Enter where appropriate, you can set or clear any bit in the matrix at random, with minimal keystroking effort. Finally, pressing Esc ends the edit session, with any changes stored in the *SetData* field of *SetObject.*

You can try out *SetObject*'s different methods by running Listing Two (page 177), SETTEST.PAS.

Pokey Video

Only two aspects of the code bear serious explaining: The way the matrix pattern is generated on the screen, and the mouse support for editing.

By sheer bad luck I was forced off of my ramcharged 25-MHz 386 and onto a gen-u-wine o-riginal 4.77-MHz IBM PC while developing *SetObject.* At first I drew the matrix on the screen using a pair of nested loops containing

(continued from page 163)
GotoXY and *Write*, as most reasonable people would. This worked fairly well on the 386. However, on the PC the results came close to making me scream: Zzzzzzzzzzzzzzzzzzzit! I could watch the matrix flow into place from the top of the screen down. Any screen that I can watch drawing itself is too damned slow; watching mainframe screens slog their way to wholeness was considered entertainment rivaling old Andy Griffith reruns back when I worked at Xerox MIS/DP.

Sorry Andy. When such things happen, I go right to assembler and a whole new approach. Instead of drawing the matrix every time I need it, I create the matrix as a memory image on the heap, and flash it in with the *Move* block move statement any time I want to display it.

Move is absurdly simple and just as fast; the trickiness is all in getting the matrix onto the heap to begin with. Using *Move* to transfer an easily-readable character array such as *MatrixText* in Listing One won't work, because a screen memory image must have an attribute byte after each and every byte to be displayed. So I wrote *MatrixBlast*, a specialized *Move* variant that moves a character array onto the heap while inserting an attribute byte after each character byte. It's an *INLINE* macro and pretty close to as fast as such a creature can be, at least until Michael Abrash gets hold of it. The set matrix I'm using here is only 16 × 80, but you can easily use *MatrixBlast* to move entire 25 × 80 screens from a text array (perhaps in the form of a typed constant) to the heap, and then flash the screen into view with *Move*.

Once moved onto the heap, the raw matrix image must be updated with highlight attributes to reflect the current state of the 255 bits in *SetData* field of *SetObject*. There's a second *INLINE* macro that does that job: *AttributeBlast*, which is called from a tight loop inside the *Show* method. *AttributeBlast* moves a specified attribute byte into the memory image on the heap, without disturbing the visible screen data already there. In a sense, *AttributeBlast* machine guns attribute bytes in between the ASCII data bytes in the image on the heap, starting at some offset from the pointer to the image, and continuing for a specified number of characters.

With *MatrixBlast* and *AttributeBlast*, there were no longer any pokey screen problems, even on the old PC.

A note on a bug that drove me bonkers: While writing *MatrixBlast*, I inadvertently entered an ordinary right parenthesis instead of a right curly bracket at the end of the first comment line:

```
{ Pop attribute character into AX)
```

The net effect was to comment out the *$5B* opcode (POP BX) on the next line, which got the stack frame all confused and blew my system away every time. It wasn't until I started tracing the program opcode-by-opcode with Turbo Debugger that the missing *$5B* became apparent, and even then it took some definite right-brain thinking. (We're not good at looking for something that isn't there!) Be careful when working with *INLINE*. The compiler lets you do what you like — and one small twitch in a bracket (or lack of one) can send your system to Bim-Bom-Bay.

Mouse Control

I'm not fond of mouse input except when it makes sense — and mouse input for intense text data entry and editing rarely makes sense, regardless of what the Mac crazies think. I type at about 110 words a minute, and if I had to break stride to grab the mouse to slide up a line or two to fix a pair of transposed letters, I'd never get anything finished on time.

Picking and toggling bits on the matrix, however, is a perfect job for the

mouse, because everything is point-and-shoot. There's no entry of characters involved; you click on a number and that bit changes state. For this reason, I built mouse control into *SetObject*'s *Edit* method. Furthermore, the user doesn't need to specify mouse control; if the mouse driver is available, *SetObject* will detect it and enable the mouse cursor. If the mouse driver isn't found, the arrow keys can be used to move the hot bit and the Enter key to change the hot bit's state.

In the *SetObj* unit's initialization section, a call is made to a function that looks for either a null or an IRET instruction (opcode $CF) and, finding neither, assumes that there is a mouse driver at the other end of interrupt vector 51. I've always thought that a little dicey, but no better way has been provided by Microsoft, so we'll run with it. A global Boolean named *MouseAvailable* is set to *True* if the unit assumes a mouse driver is installed. All through the unit, *MouseAvailable* is tested whenever something associated with the mouse needs to be done. If no driver is installed, the unit works perfectly well from the keyboard. If the mouse is there, the driver is reset and the mouse cursor turned on.

For Want of a Flag

There is a shortcoming in the Microsoft standard mouse driver that has infuriated me since I wrote my first mouse-aware program back in 1983. (I still use that original, first-run Microsoft Mouse, lint-catcher mechanical gimcrackery and all.) Whereas it's possible to turn the mouse cursor on and off at will, there's no way to determine if the mouse cursor is already on. You have to keep your own flag or make assumptions. This is dumb — there's a documented but unavailable cursor-visible flag inside the driver somewhere, and all it would take would be a little code to return it during the status function call.

I do make certain assumptions here, in that if the mouse driver has been found, the mouse cursor is assumed to be visible during the execution of the *Edit* method. This is only significant during execution of the *Show* method, which must turn off the visible mouse cursor in order to blast the matrix image from the heap into the video refresh buffer. (If you write over the mouse cursor, odd things happen.) At the end of *Show*, the mouse cursor is made visible again — if *Show* was called from within *Edit*. You don't want the mouse cursor coming on if you're just calling *Show* to display a set object's data with no intention to edit, as is done in Listing Two. Hence the otherwise worthless *EditInProcess* flag that must be carted around by all instances of *SetObj*. If we could just test to see if the mouse cursor were visible at any given time, we could just not turn it back on, if it wasn't on already when we entered *Show*. Grrrrr.

Another note on mouse control: While the mouse is visible, it is limited to travelling only within the extent of the 16 lines of the matrix, courtesy of a call to mouse function 8 just before the cursor is turned on at the top of *SetObject.Edit*. This is important; for simplicity I don't check to see if the mouse is outside the matrix, and if you let the mouse roam the screen it will still toggle the state of bits in the matrix even when outside the matrix. (Comment out the call to function 8 and you'll see what I mean.)

I suppose that only a few of you have magazines and their attendant bingo cards to service, but *SetObj* may have some other uses. Test scores? Survey results? Be creative. Consider *SetObj* a potential birdhouse, and keep your mind open to uses that don't follow from your traditional notions of what sets are for. Hey, do like I do: If it looks like a coil form — go looking for the wire!

The Last Word on Mice

Microsoft has finally published the definitive book on mouse programming,

Products Mentioned

Microsoft Mouse Programmer's Reference
(no author given)
Microsoft Press, 1989
ISBN 1-55615-191-8
Softcover, 321 pages, $29.95
Listings disk is bound into all copies

Modula-2 A Complete Guide
by K. N. King
D.C. Heath and Company, 1988
1-800-334-3284
ISBN 0-669-11091-4
Softcover, 656 pages, $36
Listings disk available from author,
Price: $10

TopSpeed Modula-2 V2.0
Jensen & Partners International
1101 San Antonio Road, Suite 301
Mountain View, CA 94043
415-967-3200
DOS Price: $199
OS/2 Price: $495

six years after it should have. Better late than never, I guess — and I'm kicking myself for not having written it back in 1985, when I had the notion. Anyway — if you have things with tails on your desktop, pick it up: *Microsoft Mouse Programmer's Reference.* (The book was written in-house and carries no author's byline.)

The book is short-ish, but it's clear, and accurate, and tolerably easy to read. There are lots of code examples, mostly in C, but the explanations are strong enough to carry you into any language that has a software interrupt primitive or allows assembly language externals. The book is expensive as thin books go, but it includes a disk with the listings and some mouse menu folderol the usefulness of which escapes me.

For the completeness of its coverage, a must-have.

More Modula Books

I've gotten a handful of new books on Modula-2 since complaining of their dearth a few columns ago. None seem much worth mentioning except one: *Modula-2 A Complete Guide,* by K.N. King. The book is by no means new (1988) but as it's a text and not a trade volume, I had no way to know of its publication before Dr. King was kind enough to send it to me.

Modula-2 A Complete Guide is by far the best book on Modula now on my shelves. It's big (650+ pages), clear, complete, and honest — it says what I discovered years ago, that Niklaus Wirth's suggested Processes module is next to worthless. Furthermore, King proposes a replacement that's considerably better, though I haven't had a chance to try it yet.

The writing rises much above the stuffy incompetence I see in far too many CS textbooks these days. Keep in mind that *Modula-2 A Complete Guide* is not a DOS book, so you won't get lots of interesting tidbits on accessing the joystick port or tweaking display adapter registers. (Good thing, too — perhaps Dr. King has left enough for me to make a book of.) You won't find it in ordinary bookstores, although a college bookstore where the book is taught would have it. Fortunately, it can be ordered directly from the publisher, through the 800 number given in the Products Mentioned box.

Go through this book and you'll have standard Modula-2 in your hip pocket. Highly recommended.

Closing In on Modula Objects

By next column I hope to have JPI's new 2.00 Version of TopSpeed Modula-2, which includes object-oriented extensions similar to those in Turbo Pascal 5.5. The siren song of objects has taken me away from Modula to a great extent this past year, and now it's time to go back, I think.

There may be some griping, especially from the portability paranoids, but hey, guys, consider: Every one of the languages I currently use is now object-oriented. Pascal, Modula-2, Smalltalk, Actor, C (no, no kidding!) so . . . what's left? The only major holdout is Basic, which I haven't touched for awhile, but there are intriguing rumblings from Microsoft that they may take Big Bill's Baby in an objective direction soon.

And then there are those ongoing rumors of object-oriented Cobol. . . .

Stranger things have happened. The Berlin Wall is gone. Democrats have started voting against tax hikes. They still grow broccoli.

In other words, don't bet the rent.

Contact Jeff Duntemann on MCI Mail as JDuntemann, or on CompuServe through ID 76117,1426.

DDJ

(Listings begin on page 172.)

Vote for your favorite feature/article.
Circle Reader Service **No. 14.**

Listing One *(Text begins on page 149.)*

```
/* ---------- Display NetWare USERID ---------- */
#include <stdio.h>
#include <nit.h>

char *GetUserid(void);

void main()
{
    printf("\nUserid: %s", GetUserid());
}

/*
 * Get the current logged on userid
 */

char *GetUserid(void)
{
    static char userid[48];
    WORD connection_number = GetConnectionNumber();
    WORD objecttype;
    long objectid;
    BYTE logintime[7];

    GetConnectionInformation(connection_number, userid,
                    &objecttype, &objectid, logintime);
    return userid;
}
```

End Listing One

Listing Two

```
/* -------- getuser1.c ----------- */

#include <stdio.h>
#include <dos.h>
#include <string.h>

/* ----- request packet for get connection information ----- */
static struct {
    int  rlen;            /* packet length-2  */
    char func;            /* NW function      */
    char station;         /* station number   */
} rqpacket = { 2, 22 };

/* ------ reply buffer for get connection information ----- */
static struct {
    int rlen;           /* packet length-2  */
    long id;            /* object id        */
    int type;           /* type of object   */
    char userid[48];    /* name of object   */
    char time[8];       /* log on time      */
} rsbuffer = { 62 };

char *GetUserid(void);

void main()
{
    printf("\nUserid: %s", GetUserid());
}

char *GetUserid(void)
{
    union REGS regs;
    struct SREGS segs;

    segread(&segs);
    segs.es = segs.ds;
    /* ------- get connection (station) number ------ */
    regs.h.ah = 0xdc;
    intdosx(&regs, &regs, &segs);
    rqpacket.station = regs.h.al;
    /* ------- get connection information --------- */
    regs.x.si = (unsigned) &rqpacket;
    regs.x.di = (unsigned) &rsbuffer;
    regs.h.ah = 0xe3;
    intdosx(&regs, &regs, &segs);
    return rsbuffer.userid;
}
```

End Listing Two

Listing Three

```
/* -------------- showusrs.c ---------------- */

#include <stdio.h>
#include <nit.h>
#include <niterror.h>

static long objid;

char *getusers(void);

void main()
{
    char *userid;
    objid = -1;
```

(Listing continued on page 170)

Listing Three *(Listing continued, text begins on page 149.)*

```
    while ((userid = getusers()) != NULL)
        printf("\n%s", userid);
}

/* ---------- scan bindery for users ------------ */
char *getusers(void)
{
    static char name[48];
    char hasproperties, flag, security;
    WORD type;

    if (ScanBinderyObject("*", OT_USER, &objid, name, &type,
              &hasproperties, &flag, &security) == SUCCESSFUL)
        return name;
    return NULL;
}
```

End Listing Three

Listing Four

```
/* --------------- nwloaded.c ------------ */

#include <stdio.h>
#include <dos.h>

int NetworkLoaded(void);

void main()
{
    printf("\nThe network %s operating",
            NetworkLoaded() ? "is" : "is not");
}

#define encr(lm) (0-(~(0-lm)))

/* ------- test network operating ----------- */
int NetworkLoaded(void)
{
    int lockmode, encrypted;
    union REGS regs;

    regs.x.ax = 0xc600;
    intdos(&regs, &regs);
    lockmode = regs.h.al;

    encrypted = encr(lockmode);

    regs.h.ah = 0xc6;
    regs.h.al = encrypted;
    intdos(&regs, &regs);
    lockmode = regs.h.al;

    return encrypted != encr(lockmode);
}
```

End Listings

Listing One *(Text begins on page 161.)*

```
{-------------------------------------------------}
{                    SETOBJ                       }
{                                                 }
{  Set object with an interactive editing method  }
{                                                 }
{                       by Jeff Duntemann         }
{                       For DDJ 8/90              }
{                       Turbo Pascal 5.5          }
{                       Last modified 5/20/90     }
{-------------------------------------------------}

UNIT SetObj;

INTERFACE

USES DOS,Crt;

TYPE
  BitSet    = SET OF 0..255;  { Maximum size generic set }

  SetObject =
    OBJECT
      SetData   : BitSet;     { The set data itself }
      HotBit    : Integer;    { Bit currently subject to editing }
      ShowAtRow : Integer;    { Matrix may appear at row 1 to 8 }
      MatrixPtr : Pointer;    { Points to matrix pattern on heap }
      Origin    : Integer;    { Display text starts at 0 or 1 }
      Attribute : Integer;    { Attribute for nonhighlighted elements }
      Highlight : Integer;    { Attribute for highlighted elements }
      EditInProcess : Boolean;    { True if inside the Edit method }
      CONSTRUCTOR Init(InitialOrigin,
                       InitialAttribute,
                       InitialHighlight,
                       InitialStartRow : Integer);
      DESTRUCTOR  Done;       { Removes object from memory }
      PROCEDURE   ClearSet;   { Forces all set bits to 0 }
      PROCEDURE   Show;       { Displays set data; doesn't edit }
      PROCEDURE   Edit;       { Displays and edits set data }
    END;


IMPLEMENTATION

TYPE
  Char40 = ARRAY[0..39] OF CHAR;  { For the matrix; see below }

CONST
  LeftCursorChar  = #16;   { These are the bracketing characters   }
  RightCursorChar = #17;   { Indicating which set element is being }
                           { edited. }

  { This is the text portion of the 16-line number matrix used to display }
  { and edit set elements.  They are first stored onto the heap, then the }
  { object's attribute is merged with the text on the heap.  This way,    }
  { you can move the whole image onto the screen, attributes and all,     }
  { with a single Move statement. }

  MatrixText : ARRAY[0..32] OF Char40 = (
           ' 000  001  002  003  004  005  006  007 ',
           ' 008  009  010  011  012  013  014  015 ',
           ' 016  017  018  019  020  021  022  023 ',
           ' 024  025  026  027  028  029  030  031 ',
           ' 032  033  034  035  036  037  038  039 ',
           ' 040  041  042  043  044  045  046  047 ',
           ' 048  049  050  051  052  053  054  055 ',
           ' 056  057  058  059  060  061  062  063 ',
           ' 064  065  066  067  068  069  070  071 ',
           ' 072  073  074  075  076  077  078  079 ',
           ' 080  081  082  083  084  085  086  087 ',
           ' 088  089  090  091  092  093  094  095 ',
           ' 096  097  098  099  100  101  102  103 ',
           ' 104  105  106  107  108  109  110  111 ',
           ' 112  113  114  115  116  117  118  119 ',
           ' 120  121  122  123  124  125  126  127 ',
           ' 128  129  130  131  132  133  134  135 ',
           ' 136  137  138  139  140  141  142  143 ',
           ' 144  145  146  147  148  149  150  151 ',
           ' 152  153  154  155  156  157  158  159 ',
           ' 160  161  162  163  164  165  166  167 ',
           ' 168  169  170  171  172  173  174  175 ',
           ' 176  177  178  179  180  181  182  183 ',
           ' 184  185  186  187  188  189  190  191 ',
           ' 192  193  194  195  196  197  198  199 ',
           ' 200  201  202  203  204  205  206  207 ',
           ' 208  209  210  211  212  213  214  215 ',
           ' 216  217  218  219  220  221  222  223 ',
           ' 224  225  226  227  228  229  230  231 ',
           ' 232  233  224  235  236  237  238  239 ',
           ' 240  241  242  243  244  245  246  247 ',
           ' 248  249  250  251  252  253  254  255 ',
           ' 256                                   ');

VAR
  VidBufferPtr   : Pointer;  { Global, set in the init. section }
  MouseAvailable : Boolean;  { Global, set in the init. section }

{-------------------------------------------------}
{  Procedures and functions private to this unit: }
{-------------------------------------------------}


{ This is the general-purpose mouse call primitive: }

PROCEDURE MouseCall(VAR M1,M2,M3,M4 : Word);

VAR
  Regs : Registers;

BEGIN
  WITH Regs DO
    BEGIN
      AX := M1; BX := M2; CX := M3; DX := M4;
    END;
  INTR(51,Regs);  { 51 = $33 = Mouse driver interrupt vector }
  WITH Regs DO
    BEGIN
      M1 := AX; M2 := BX; M3 := CX; M4 := DX;
    END;
END;


PROCEDURE ShowMouse;

VAR
  M1,M2,M3,M4 : Word;

BEGIN
  M1 := 1; MouseCall(M1,M2,M3,M4);
END;


PROCEDURE HideMouse;

VAR
  M1,M2,M3,M4 : Word;

BEGIN
  M1 := 2; MouseCall(M1,M2,M3,M4);
END;


{ If called when left mouse button is down, waits for release }

PROCEDURE WaitForMouseRelease;

VAR
  M1,ButtonStatus,M3,M4 : Word;

BEGIN
  M1 := 3;
  REPEAT
    MouseCall(M1,ButtonStatus,M3,M4);
  UNTIL NOT Odd(ButtonStatus);  { Wait until Bit 0 goes to 0 }
END;



PROCEDURE UhUh;  { Says "uh-uh" when you press the wrong key }

VAR
  I : Integer;

BEGIN
  FOR I := 1 TO 2 DO
    BEGIN
      Sound(50); Delay(100); NoSound; Delay(50);
    END;
END;



FUNCTION MouseIsInstalled : Boolean;

TYPE
  BytePtr = ^Byte;

VAR
  TestVector : BytePtr;

BEGIN
  GetIntVec(51,Pointer(TestVector));
  { $CF is the binary opcode for the IRET instruction; }
  { in many BIOSes, the startup code puts IRETs into   }
  { most unused bectors. NIL, of course, is 4 zeroes.  }
  IF (TestVector = NIL) OR (TestVector^ = $CF) THEN
    MouseIsInstalled := False
  ELSE
    MouseIsInstalled := True
END;



{ Returns True if running on a mono system: }

FUNCTION IsMono : Boolean;

VAR
  Regs : Registers;
BEGIN
  Intr(17,Regs);
  IF (Regs.AX AND $0030) = $30 THEN IsMono := True
    ELSE IsMono := False;
END;


{-------------------------------------------------------}
{ Returns True if left mouse button was clicked, and if }
{ the button *was* clicked, returns the X,Y position    }
{ of the mouse at click-time in MouseX,MouseY.  If      }
{ called when the mouse was *not* clicked, returns 0    }
{ in MouseX and MouseY.                                 }
{-------------------------------------------------------}

FUNCTION MouseWasClicked(VAR MouseX,MouseY : Word) : Boolean;

VAR
  M1,ButtonStatus : Word;

BEGIN
```

(Listing continued on page 174)

Listing One *(Listing continued, text begins on page 161.)*

```
  M1 := 3; MouseCall(M1,ButtonStatus,MouseX,MouseY);
  IF Odd(ButtonStatus) THEN MouseWasClicked := True
    ELSE
      BEGIN
        MouseWasClicked := False;
        MouseX := 0;
        MouseY := 0;
      END
END;


PROCEDURE MatrixBlast(TextPtr,Heapptr  : Pointer;
                      SizeOfMatrix     : Word;
                      Origin,Attribute : Byte);

INLINE
($58/     { POP AX }      { Pop attribute character into AX}
 $5B/     { POP BX }      { Pop origin digit into BX }
 $59/     { POP CX }      { Pop byte count into CX }
 $5F/     { POP DI }      { Pop heap pointer offset portion into DI }
 $07/     { POP ES }      { Pop heap pointer segment portion into ES }
 $5E/     { POP SI }      { Pop matrix pointer offset portion into SI }
 $5A/     { POP DX }      { Pop matrix pointer segment portion into DX }
 $1E/     { PUSH DS }     { Store Turbo's DS value on the stack }
 $8E/$DA/ { MOV DS,DX }   { Move DX into DS }
 $86/$C4/ { XCHG AL,AH }  { Get attribute into hi byte of AX }
 $03/$F3/ { ADD SI,BX }   { Add origin adj. to matrix pointer offset }
 $AC/     { LODSB }       { Load MatrixText character at DS:SI into AL }
 $AB/     { STOSW }       { Store matrix char/attr pair in AX to ES:DI }
 $E2/$FC/ { LOOP -4 }     { Loop back to LOADSB until CX = 0 }
 $1F);    { POP DS }      { Pop Turbo's DS value from stack back to DS }


 PROCEDURE AttributeBlast(ImagePtr : Pointer;
                          ImageOffset,Attribute,WordCount : Integer);

 INLINE(
 $59/     { POP CX }      { Pop word count into CX }
 $58/     { POP AX }      { Pop attribute value into AX }
 $5B/     { POP BX }      { Pop image offset value into BX }
 $D1/$E3/ { SHL BX,1 }    { Multiply image offset by 2, for words not bytes }
 $5F/     { POP DI }      { Pop offset portion of image pointer into DI }
 $07/     { POP ES }      { Pop segment portion of image pointer into ES }
 $03/$FB/ { ADD DI,BX }   { Add image offset value to pointer offset }
 $47/     { INC DI }      { Add 1 to DI to point to attribute of 1st char }
 $AA/     { STOSB }       { Store AL to ES:DI; INC DI by 1 }
 $47/     { INC DI }      { Increment DI past character byte }
 $E2/$FC);{ LOOP -4 }     { Loop back to STOSB until CX = 0 }


{-------------------------------------}
{  Method definitions for SetObject: }
{-------------------------------------}

CONSTRUCTOR SetObject.Init(InitialOrigin,
                           InitialAttribute,
                           InitialHighlight,
                           InitialStartRow : Integer);

BEGIN
  { Set initial values for state variables: }
  Origin    := InitialOrigin;
  Attribute := InitialAttribute;
  Highlight := InitialHighlight;
  ShowAtRow := InitialStartRow;
  SetData   := [];         { Set initial set value to empty }
  HotBit    := 0;          { Set initial hot bit to 0 }
  EditInProcess := False; { Not in Edit method right now! }

  { Allocate space on the heap for the matrix: }
  GetMem(MatrixPtr,SizeOf(MatrixText)*2);
  { Blast the matrix pattern, with attributes, onto the heap: }
  MatrixBlast(@MatrixText,MatrixPtr,
              SizeOf(MatrixText),(Origin*5),Attribute);
END;


DESTRUCTOR SetObject.Done;

BEGIN
  { Free the memory occupied by the matrix image: }
  FreeMem(MatrixPtr,SizeOf(MatrixText)*2);
END;


PROCEDURE SetObject.ClearSet;

BEGIN
  FillChar(SetData,Sizeof(SetData),Chr(0));
END;



PROCEDURE SetObject.Show;

VAR
  I,Offset : Integer;
  ShowPtr  : Pointer;

BEGIN
  { It's important not to clobber the visible mouse cursor in the }
  { video refresh buffer.  This is why we turn it off for the     }
  { duration of this procedure: }
  IF MouseAvailable THEN IF EditInProcess THEN HideMouse;
  FOR I := 0 TO 255 DO
    IF I IN SetData THEN
      AttributeBlast(MatrixPtr,(I*5)+1,Highlight,3)
    ELSE
```

```
      AttributeBlast(MatrixPtr,(I*5)+1,Attribute,3);
  Offset := (ShowAtRow-1) * 160;  { Offset in bytes into the vid. buffer }
  { Create a pointer to the matrix location in the video buffer: }
  ShowPtr := Pointer(LongInt(VidBufferPtr) + Offset);
  { Move the matrix image from the heap into the video buffer: }
  Move(MatrixPtr^,ShowPtr^,(Sizeof(MatrixText) SHL 1)-79);
  { If the mouse is available we assume we're using it: }
  IF MouseAvailable THEN IF EditInProcess THEN ShowMouse;
END;


{-------------------------------------------------------------------}
{ This is the beef of the SetObject concept: A method that brings up }
{ a 16 X 16 matrix of bit numbers, each of which corrresponds to one }
{ bit in the set.  The method allows the user to zero in on a single }
{ bit through the keyboard or through the mouse if the driver is     }
{ loaded. Click on the number (or press Enter) and the bit changes   }
{ state, as indicated by screen highlighting.  This is useful for    }
{ debugging or even data entry to a set object.                      }
{-------------------------------------------------------------------}

PROCEDURE SetObject.Edit;

VAR
  I             : Integer;
  M1,M2,M3,M4   : Word;
  MouseX,MouseY : Word;
  Quit          : Boolean;
  InCh          : Char;


PROCEDURE PokeToCursor(Left,Right : Char);

BEGIN
  Char(Pointer(LongInt(MatrixPtr)+(HotBit*10))^)   := Left;
  Char(Pointer(LongInt(MatrixPtr)+(HotBit*10)+8)^) := Right;
END;


PROCEDURE MoveHotBitTo(NewHotBit : Integer);

BEGIN
  PokeToCursor(' ',' ');
  HotBit := NewHotBit;
  PokeToCursor(LeftCursorChar,RightCursorChar);
  Show;
END;


{ Converts a mouse screen X,Y to a bit position in the matrix }
{ from 0-255: }

FUNCTION MouseBitPosition(MouseX,MouseY : Integer) : Integer;

VAR
  ScreenX,ScreenY : Word;

BEGIN
  ScreenX := (MouseX DIV 8) + 1; ScreenY := (MouseY DIV 8) + 1;
  ScreenY := ScreenY - ShowAtRow;  { Adjust Y for screen position }
  MouseBitPosition := (ScreenY * 16) + (ScreenX DIV 5);
END;


{ Simply toggles the set bit specified in FlipBitNumber: }

PROCEDURE ToggleBit(FlipBitNumber : Integer);

BEGIN
  IF FlipBitNumber IN SetData THEN  { If it's a 1-bit }
    BEGIN
      SetData := SetData - [FlipBitNumber];
      AttributeBlast(MatrixPtr,(FlipBitNumber*5)+1,Attribute,3);
    END
  ELSE   { If it's a 0-bit }
    SetData := SetData + [FlipBitNumber];
END;


BEGIN  { Body of Edit }
  EditInProcess := True;
  { Make keyboard cursor visible at HotBit: }
  PokeToCursor(LeftCursorChar,RightCursorChar);
  Show;

  { Turn on mouse cursor if mouse is available: }
  IF MouseAvailable THEN
    BEGIN
      M1 := 0; MouseCall(M1,M2,M3,M4); { Reset mouse }
      M1 := 8; M3 := ((ShowAtRow-1) SHL 3);
      M4 := ((ShowAtRow-1) SHL 3) + 120;
      MouseCall(M1,M2,M3,M4);  { Limit mouse movement vertically }
      M1 := 1; MouseCall(M1,M2,M3,M4); { Show mouse cursor }
    END;

  Quit := False;
  REPEAT
    IF MouseAvailable THEN     { Test global Boolean variable }
      IF MouseWasClicked(MouseX,MouseY) THEN
        BEGIN                          { Mouse was clicked... }
          I := MouseBitPosition(MouseX,MouseY); {..on what bit? }
          MoveHotBitTo(I);           { Move hot bit to that bit }
          ToggleBit(I);       { Toggle the selected bit's state }
          WaitForMouseRelease;        { Wait for button release }
          Show;                          { Redisplay the matrix }
        END;

    IF KeyPressed THEN  { If the user pressed any key... }
```

(Listing continued on page 177)

Listing One *(Listing continued, text begins on page 161.)*

```
      BEGIN
        InCh := ReadKey;          { Get the key }
        IF InCh = Chr(0) THEN     { If it was null... }
          BEGIN
            InCh := ReadKey;      { Get the second half }
            CASE Ord(InCh) OF     {  and parse it: }
  { Up }      72 : IF HotBit > 15 THEN I := HotBit-16 ELSE Uhuh;
  { Left }    75 : IF HotBit > 0 THEN I := Hotbit-1 ELSE Uhuh;
  { Right }   77 : IF HotBit < 255 THEN I := HotBit+1 ELSE Uhuh;
  { Down }    80 : IF HotBit < 239 THEN I := HotBit+16 ELSE Uhuh;
  { Home }    71 : I := 0;
  { PgUp }    73 : I := 15;
  { End }     79 : I := 240;
  { PgDn }    81 : I := 255;
              ELSE Uhuh;
            END; { CASE }
            MoveHotBitTo(I);
          END;
        CASE Ord(InCh) OF
          13 : ToggleBit(HotBit);     { Enter }
          27 : Quit := True;          { ESC }
          ELSE {Uhuh;}
        END; { CASE }
        Show;
      END;
  UNTIL Quit;
  IF MouseAvailable THEN HideMouse;      { Hide mouse cursor }
  PokeToCursor(' ',' '); { Erase cursor framing characters }
  EditInProcess := False;
END;


{ Initialization section: }

BEGIN
  IF IsMono THEN VidBufferPtr := Ptr($B000,0)
    ELSE VidBufferPtr := Ptr($B800,0);
  { Here we look for the presence of the mouse driver: }
  MouseAvailable := MouseIsInstalled;
END.
```

End Listing One

Listing Two

```
PROGRAM SetTest;

USES Crt,SetObj;  { SetObj presented in DDJ 8/90 }

VAR
  MySet : SetObject;

BEGIN
  TextBackground(Black);
  ClrScr;
  MySet.Init(0,$07,$70,1);                { Create the object }
  MySet.SetData := [0,17,42,121,93,250]; { Give set a value }
  MySet.Edit;                             { Edit the set }
  ClrScr;                                 { Clear screen }
  Readln;                                 { Wait for keypress }
  MySet.Show;                             { Show the set }
  MySet.ClearSet;                         { Zero the set }
  Readln;                                 { Wait for keypress }
  MySet.Show;                             { Show the cleared set }
  Readln;                          { And wait for final keypress }
END.
```

End Listings

Listing One *(Text begins on page 80.)*

```
/* bad.c -- Mistakes in memory allocation */

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

main()
{
   char *allocated_but_never_freed;
   char *this_one_is_ok;
   char *freed_but_never_allocated;

   allocated_but_never_freed = malloc(10);
   this_one_is_ok            = malloc(20);

   free(this_one_is_ok);
   free(freed_but_never_allocated);

   return(0);
}
```

End Listing One

Listing Two

```
/* bad.c -- Mistakes in memory allocation */

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

#include <mem.h>

main()
{
   char *allocated_but_never_freed;
   char *this_one_is_ok;
   char *freed_but_never_allocated;

   allocated_but_never_freed = memMalloc(10,"tag 1");
   this_one_is_ok            = memMalloc(20,"tag 2");

   memFree(this_one_is_ok,           "tag 3");
   memFree(freed_but_never_allocated,"tag 4");

   return(0);
}
```

End Listing Two

Listing Three

```
/* memMalloc() -- Same as malloc(), but registers activity using memTrack().
* Copyright (c) 1990, Cornerstone Systems Group, Inc.
*/

#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>

#include <mem.h>

void *memMalloc(size_t bytes, char *tag)
{
   void *allocated;
   allocated = malloc(bytes);
   memTrack_alloc(allocated, tag);
   return(allocated);
}

/* memFree() -- Same as free(), but registers activity using memTrack().
*  Copyright (c) 1990, Cornerstone Systems Group, Inc.
*/

#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>

#include <mem.h>

void memFree(void *to_free, char *tag)
{
   if (memTrack_free(to_free, tag))
      {
      free(to_free);
      }
}
/* MEMTRACK.C -- Module to track memory allocations and frees that occur
* in the other mem...() routines. Global routines:
*     memTrack_alloc() -- Records allocations.
*     memTrack_free()  -- Records attempts to free.
*  Copyright (c) 1990, Cornerstone Systems Group, Inc.
*/

#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>

#include <mem.h>
static FILE  *memTrack_fp(void);
static void   memTrack_msg(char *msg);
```

```
#define  ALLOC   'A'
#define  FREE    'F'

/* Track an allocation. Write it in the debugging file in the format
*   A 0000:0000 tag */
void memTrack_alloc(void *allocated, char *tag)
{
   FILE  *fp;

   if (fp = memTrack_fp())
      {
      fseek(fp,0L,SEEK_END);
      fprintf(fp,"%c %p %s\n",ALLOC, allocated, tag);
      fclose(fp);
      }
}

/*  Track freeing of pointer. Return FALSE if was not allocated, but tracking
 *  file exists. Return TRUE otherwise. */
int memTrack_free(void *to_free, char *tag)
{
   int   rc = 1;
   FILE  *fp;
   void  *addr_in_file = 0;
#define  MAX_LTH  200
   char  line[MAX_LTH];
   char  found = 0;
   char  dummy;
   int   ii;
   long  loc;
   if (fp = memTrack_fp())
      {
      rewind(fp);
      for ( loc=0L; fgets(line,MAX_LTH,fp); loc = ftell(fp) )
         {
         if (line[0] != ALLOC)         /* Is the line an 'Allocated' line?  */
            continue;                  /*   If not, back to top of loop.    */
         ii = sscanf(line,"%c %p",&dummy, &addr_in_file);
         if (ii==0 || ii==EOF)
            continue;
                                       /* Is addr in file the one we want?  */
         if ( (char *)addr_in_file - (char *)to_free == 0 )
            {
            found = 1;
            fseek(fp,loc,SEEK_SET);    /* Back to start of line     */
            fputc(FREE,fp);            /* Over-write the ALLOC tag */
            break;
            }
         }
      fclose(fp);
      if (!found)
         {
         char  msg[80];
         sprintf(msg,"Tried to free %p (%s).  Not allocated.",to_free,tag);
         memTrack_msg(msg);
         }
      }
   return(rc);
}

/* Return FILE pointer for tracking file.  */
static FILE  *memTrack_fp()
{
   static char  *ep = NULL;    /* Points to environment var that names file */
   FILE  *fp = NULL;           /* File pointer to return                    */

   if (ep == NULL              /* First time through, just create blank file */
   &&   (ep = getenv("MEMTRACK"))
   &&   (fp = fopen(ep,"w")) )
      {
      fclose(fp);
      fp = 0;
      }
   if (ep)                     /* If we have a file name, proceed.          */
      {                        /*   Otherwise, do nothing.                  */
      fp = fopen(ep,"r+");     /* Open debugging file for append access.    */
      if (!fp)
         {
         fprintf(stderr,"\a\nCannot open %s\n\a",ep);
         }
      }
   return(fp);
}

/* Write a message to the debugging file. */
static void memTrack_msg(char *msg)
{
   FILE  *fp;

   if (fp = memTrack_fp())
      {
      fseek(fp,0L,SEEK_END);
      fprintf(fp,"\n%s\n",msg);
      fclose(fp);
      }
   else
      {
      fprintf(stderr,"%s\n",msg);

      }
}

/* memCalloc() -- Same as calloc(), but registers activity using memTrack().
*  Copyright (c) 1990, Cornerstone Systems Group, Inc.
*/
#include <stdlib.h>
#include <stdio.h>
```

(Listing continued on page 180)

Listing Three *(Listing continued, text begins on page 80.)*

```
#include <malloc.h>
#include <mem.h>

void *memCalloc(size_t num_elems, size_t bytes_per_elem, char *tag)
{
   void *allocated;
   allocated = calloc(num_elems, bytes_per_elem);
   memTrack_alloc(allocated, tag);
   return(allocated);
}

/* memRealloc() - Same as realloc(), but registers activity with memTrack().
*  Copyright (c) 1990, Cornerstone Systems Group, Inc.
*/
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>
#include <mem.h>

void *memRealloc(void *allocated, size_t bytes, char *tag)
{
   memTrack_free(allocated, tag);
   allocated = realloc(allocated, bytes);
   if (allocated)
      {
      memTrack_alloc(allocated, tag);
      }
   return(allocated);
}

/* memStrdup() -- Same as strdup(), but registers activity using memTrack().
*  Copyright (c) 1990, Cornerstone Systems Group, Inc.
*/
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>
#include <string.h>
#include <mem.h>

void *memStrdup(void *string, char *tag)
{
   void *allocated;
   allocated = strdup(string);
   memTrack_alloc(allocated, tag);
   return(allocated);
}
```

End Listing Three

Listing Four

```
/* MEM.H  -- ** Copyright (c) 1990, Cornerstone Systems Group, Inc. */

#ifdef MEMTRACK

void *memCalloc(size_t num_elems, size_t bytes_per_elem, char *tag);
void  memFree(void *vp, char *tag);
void *memMalloc(size_t bytes, char *tag);
void *memRealloc(void *oldloc, size_t newbytes, char *tag);
void *memStrdup(void *string, char *tag);
     /* The next two functions are only called by the other mem functions   */
void memTrack_alloc(void *vp, char *tag);
int  memTrack_free(void *vp, char *tag);
#else
#define  memCalloc(NUM,BYTES_EACH,TAG)        calloc(NUM,BYTES_EACH)
#define  memFree(POINTER,TAG)                 free(POINTER)
#define  memMalloc(BYTES,TAG)                 malloc(BYTES)
#define  memRealloc(OLD_POINTER,BYTES,TAG)    realloc(OLD_POINTER,BYTES)
#define  memStrdup(STRING, TAG)               strdup(STRING)
#endif
```

End Listing Four

Listing Five

```
/* DEMOHEAP.C - Demonstrate use of heap...() functions.
*  Copyright (c) 1990 - Cornerstone Systems Group, Inc.
*/

#include <stdio.h>
#include <malloc.h>
#include <heap.h>

static void my_own_msg_func(char *msg);

main()
{
   char *allocated_but_never_freed;
   char *this_one_is_ok;
   char *freed_but_never_allocated;
   heapPrt_set_msg_func(my_own_msg_func);
   allocated_but_never_freed = malloc(10);
   heapPrt("after first malloc()");
   this_one_is_ok            = malloc(20);
   heapPrt("after second malloc()");
   free(this_one_is_ok);
   heapPrt("after first free()");
   free(freed_but_never_allocated);
   heapPrt("after second free()");
   return(0);
}

/* heapPrt() makes its report with puts() by default.  This will not be
*  appropriate for some applications, so we will demonstrate the use of an
*  alternative message function.  This one writes to stderr.
*  The alternative function should take one argument (a char *).  Its
*  return value is ignored, so it might as well be void.
*/
static void my_own_msg_func(char *msg)
{
   fprintf(stderr,"My own message function: %s\n",msg);
}
OUTPUT:
My own message function:     1 allocations,     10 bytes, after first malloc()
My own message function:     2 allocations,     30 bytes, after second malloc()
My own message function:     1 allocations,     10 bytes, after first free()
My own message function:     1 allocations,     10 bytes, after second free()
```

End Listing Five

Listing Six

```
/* heap.h - Header file for use with heap...() functions.
*   Copyright (c) 1990 - Cornerstone Systems Group, Inc.
*/

void heapPrt(char *tag);
void heapPrt_set_msg_func(void (*new_msg_func)() );
void heapUsed(unsigned int *numused, long *totbytes);

/* HEAPUSED.C -- Tell how much of heap has been used. For use with MS C 5.x
*  Copyright (c) 1990, Cornerstone Systems Group, Inc.
*/
#include <malloc.h>
#include <heap.h>

void heapUsed(
unsigned int   *numused,
long           *totbytes)
{
   struct _heapinfo hinfo;
   int    status;
   *numused  = 0;
   *totbytes = 0L;
   hinfo._pentry = (char *)0;
   while ( (status=_heapwalk(&hinfo)) == _HEAPOK)
      {
      if (hinfo._useflag == _USEDENTRY)
         {
         ++ (*numused);
         *totbytes += hinfo._size;
         }
      }
}

/* HEAPPRT.C -- Print summary information about heap. For use with MS C 5.x
*   This module contains two functions:
*     heapPrt() prints the summary information.
*     heapPrt_set_msg_func() allows you to specify a function for heapPrt()
*        to use, other than printf().
*   Copyright (c) 1990, Cornerstone Systems Group, Inc.
*/
#include <stdio.h>
#include <malloc.h>
#include <heap.h>
static void (*heapPrt_msg_func)() = 0;

/*-------------------------------------------------------------------------*/
void heapPrt(
char  *tag)        /* Description of where you are in processing           */
{
   unsigned int   numused;          /* Number of allocations used          */
   long           totbytes;         /* Total bytes allocated               */
   char           msg[80];          /* Message to display                  */
   heapUsed(&numused, &totbytes);
   if (!heapPrt_msg_func)
      heapPrt_msg_func = puts;
   sprintf(msg, "%5u allocations, %6ld bytes, %s",numused,totbytes,tag);
   heapPrt_msg_func(msg);
}
/*-------------------------------------------------------------------------*/
void heapPrt_set_msg_func(
void (*new_msg_func)())
{
   heapPrt_msg_func = new_msg_func;
}
/*-------------------------------------------------------------------------*/
```

End Listings

Listing One *(Text begins on page 134.)*

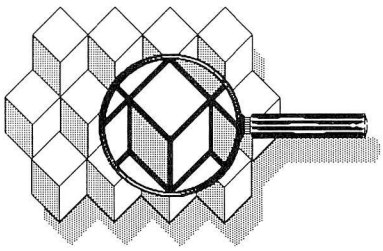
```
/* OS2ERR.H version 1.0, March, 1989 */
/* Include this file after including OS2.H or OS2DEF.H */

#ifdef OS2ERR
    USHORT APIENTRY xos2chk(PSZ, USHORT, PSZ, USHORT);
    USHORT APIENTRY xos2(PSZ, USHORT, PSZ, USHORT);
    VOID   APIENTRY xpoperr(VOID);
    #define os2chk(ErrCode) (xos2chk(__FILE__, __LINE__, #ErrCode, ErrCode))
    #define os2(ErrCode) (xos2(__FILE__, __LINE__, #ErrCode, ErrCode))
    #define poperr() (xpoperr())
#else
    #define os2chk(ErrCode) (ErrCode)
    #define os2(ErrCode) (ErrCode)
    #define poperr()
#endif
```

End Listing One

Listing Two

```
/* OS2ERR.C version 1.1, April 1989 by Nico Mak */
/* These functions are called by the macros in OS2ERR.H */
/* Compile this program with the options you use for the rest of */
/* your application, and link your application with OS2ERR.OBJ */

#define MAX_SAVE_ENTRIES 7 /* maximum number of threads that will use OS2ERR */

#include <stdio.h>      /* definitions/declarations for standard I/O routines */
#define INCL_BASE       /* to include all of OS/2 base */
#include "os2def.h"     /* include OS/2 common definitions */
#include "bse.h"        /* include OS/2 base definitions */
#define OS2ERR          /* to include OS2ERR declarations and macros */
#include "os2err.h"     /* include OS2ERR declarations */

typedef struct _SAVEINFO {                  /* save */
    TID tid;
    PSZ pszFileName;
    USHORT usLineNumber;
    PSZ pszLineSource;
    USHORT usErrCode;
    } SAVEINFO, FAR *PSAVEINFO;

SAVEINFO save[MAX_SAVE_ENTRIES];
USHORT cSavedEntries = 0;   /* number of threads that have used OS2ERR */
BOOL fOverFlow = 0;         /* 1 when all entries in "save" table are full */

/* DosErrClass error classifications */
PSZ pszClass[] = {
    "",
    "Out of Resource",
    "Temporary Situation",
    "Permission problem",
    "Internal System Error",
    "Hardware Failure",
    "System Failure",
    "Application Error",
    "Not Found",
    "Bad Format",
    "Locked",
    "Media Failure",
    "Collision with Existing Item",
    "Unknown/other",
    "Can't perform requested action",
    "Time-out",
    };

/* DosErrClass recommended actions */
PSZ pszAction[] = {
    "",
    "Retry immediately",
    "Delay and retry",
    "User error, get new values",
    "Abort in orderly manner",
    "Abort immediately",
    "Ignore this error",
    "Retry after user intervention",
    };

/* DosErrClass locus */
PSZ pszLocus[] = {
    "",
    "Unknown",
    "Disk",
    "Network",
    "Serial device",
    "Memory parameter",
    };

CHAR szWaitMsg[] = "\nPress Esc to abort process or any other key to continue";
CHAR szErrTID[] = "\npoperr() error: no information saved for this thread\n\r";
CHAR szErrOverFlow[] = "\npoperr() error: thread storage area overflowed\n\r";
CHAR szBuffer[400];

USHORT PASCAL PszLen(PSZ psz); /* function returns length of a far string */
PSAVEINFO PASCAL FindSaveEntry(TID); /* function returns PSAVEINFO for TID */

/* xos2chk - handle error (if any) and return error code */
USHORT APIENTRY xos2chk(PSZ pszFileName, USHORT usLineNumber,
 PSZ pszLineSource, USHORT usErrCode)
    {
    if (xos2(pszFileName, usLineNumber, pszLineSource, usErrCode))
        xpoperr();
    return usErrCode;
    }

/* xos2 - save error information and return error code */
USHORT APIENTRY xos2(PSZ pszFileName, USHORT usLineNumber,
 PSZ pszLineSource, USHORT usErrCode)
    {
    PSAVEINFO psave;
    PIDINFO pidi;

    if (usErrCode)
        {
        DosGetPID(&pidi);
        if ((psave = FindSaveEntry(pidi.tid)) == NULL)
            {
            DosEnterCritSec();
            if (cSavedEntries < MAX_SAVE_ENTRIES)
                psave = save + cSavedEntries++;
            else
                fOverFlow = 1;
            DosExitCritSec();
            }
        if (psave)
            {
            psave->tid = pidi.tid;
            psave->pszFileName = pszFileName;
            psave->usLineNumber = usLineNumber;
            psave->pszLineSource = pszLineSource;
            psave->usErrCode = usErrCode;
            }
        }
    return usErrCode;
    }

/* xpoperr - display pop-up screen with error information, return error code */
VOID APIENTRY xpoperr(VOID)
    {
    KBDKEYINFO kbci;
    PIDINFO pidi;
    PSZ psz;
    USHORT usClass, usAction, usLocus, usWait, usEnviron, usOffsetCmd, cbMsg;
    PSAVEINFO psave;

    /* open a pop-up screen */
    usWait = VP_WAIT | VP_OPAQUE;
    VioPopUp(&usWait, 0);

    DosGetPID(&pidi);
    if ((psave = FindSaveEntry(pidi.tid)) == NULL)
        {
        VioWrtTTY(szErrTID, sizeof(szErrTID), 0);
        if (fOverFlow)
            VioWrtTTY(szErrOverFlow, sizeof(szErrOverFlow), 0);
        }
    else
        {
        /* display error code, command name, and command tail */
        sprintf(szBuffer, "Error Code:   %u\n\r", psave->usErrCode);
        VioWrtTTY(szBuffer, PszLen(szBuffer), 0);
        DosGetEnv(&usEnviron, &usOffsetCmd);
        for (psz = MAKEP(usEnviron, 0); *psz; psz += PszLen(psz) + 1)
           ;
        psz += 1;
        sprintf(szBuffer, "Command:      %Fs\n\r", psz);
        VioWrtTTY(szBuffer, PszLen(szBuffer), 0);
        psz += PszLen(psz) + 1;
        psz += PszLen(psz) + 1;
        sprintf(szBuffer, "Command tail: %Fs\n\r", psz);
        VioWrtTTY(szBuffer, PszLen(szBuffer), 0);

        /* display process id, thread id, and DosErrClass information */
        sprintf(szBuffer, "Process ID:   %u\n\rThread ID:    %u\n\r",
         pidi.pid, pidi.tid);
        VioWrtTTY(szBuffer, PszLen(szBuffer), 0);
        DosErrClass(psave->usErrCode, &usClass, &usAction, &usLocus);
        sprintf(szBuffer,
         "Action:       %Fs\n\rLocus:        %Fs\n\rClass:        %Fs\n\r",
         pszAction[usAction], pszLocus[usLocus], pszClass[usClass]);
        VioWrtTTY(szBuffer, PszLen(szBuffer), 0);

        /* display source filename, line number, and source code */
        sprintf(szBuffer, "Source name: %Fs\n\rSource line: %d (%Fs)\n\r\n\r",
         psave->pszFileName, psave->usLineNumber, psave->pszLineSource);
        VioWrtTTY(szBuffer, PszLen(szBuffer), 0);

        /* display system message (if any) */
        if (!DosGetMessage(0, 0, szBuffer, sizeof(szBuffer), psave->usErrCode,
          "OSO001.MSG", &cbMsg))
            VioWrtTTY(szBuffer, cbMsg, 0);
        }

    /* wait for a keypress */
    VioWrtTTY(szWaitMsg, sizeof(szWaitMsg), 0);
    do {
        KbdCharIn(&kbci, IO_WAIT, 0);
        } while (!(kbci.fbStatus & 0x40));

    /* close the pop-up screen and abort if requested */
    VioEndPopUp(0);
    if (kbci.chChar == 27 && kbci.chScan == 1)
        DosExit(EXIT_PROCESS, 1);
    }

/* PszLen - return length of a far string */
USHORT PASCAL PszLen(PSZ psz)
    {
    PSZ pszSearch = psz;

    while (*pszSearch)
        pszSearch++;
    return pszSearch - psz;
    }

/* FindSaveEntry - return pointer to SAVEINFO for a thread ID */
PSAVEINFO PASCAL FindSaveEntry(TID tid)
    {
    PSAVEINFO psave;

    for (psave = save; psave < save + cSavedEntries; ++psave)
        if (psave->tid == tid)
            return psave;
    return NULL;
    }
```

End Listings



Version 5.0 of DR DOS is available from **Digital Research**. The DR DOS operating system provides MemoryMAX, a memory management facility which moves the operating system, TSRs, buffers, and drivers (including network drivers) into high memory. MemoryMAX automatically configures itself for the system in use (286, 386, or 486); users no longer have to unload networking software in order to run large applications.

ViewMAX is a Common User Access (CUA) keystroke-compatible GUI that is designed for use with a keyboard or a mouse. Users can view their disk layout as icons, text, or in a tree format. ViewMAX also supports password-protected files and subdirectories.

FileLINK is a file transfer utility that installs itself from one machine to the other via the serial link, which enables DR DOS 5.0-based systems to communicate with machines running regular DOS.

Other features include disk-caching for improving application throughput and BatteryMAX for battery longevity on portable systems. The company believes DR DOS will continue to extend the viability of DOS well into the 1990s. DR DOS retails for $199. Reader service no. 22.
Digital Research Inc.
Box DRI
70 Garden Ct.
Monterey, CA 93942
800-443-4200
408-649-3896

An industry-wide committee, including Borland, Eclipse, IGC, Intel, Locus, Lotus Development, Microsoft, Phar Lap, Phoenix Technologies, Quarterdeck, and Rational Systems, has defined a standard interface that allows extended DOS applications to take advantage of the capabilities of protected-mode, multitasking operating environments for PCs equipped with Intel's 286, 386, and i486 microprocessors. The **DOS Protected Mode Interface (DPMI)** goes beyond the Virtual Control Program Interface (VCPI) developed in 1987 by Phar Lap and Quarterdeck, which allowed expanded memory managers and DOS extenders to coexist, but did not address multitasking.

The DPMI provides reliable multitasking under a variety of environments, including those supporting system-wide virtual memory, as well as binary portability for extended DOS applications across multiple operating environments able to run DOS. The API calls defined in the DPMI spec allow DOS extenders to run on any operating system or control program on 386 and i486 systems. Existing DOS extenders can add DPMI services to their applications in addition to existing stand-alone DOS and VCPI support, permitting maximum interoperability of extended DOS applications among systems based on the Intel386 Architecture. A 386 control program or operating system that supports the DPMI spec can also support DOS extender applications that follow the standard. DPMI will be implemented by DESQview, Microsoft Windows, OS/2, Unix 386, and VM/386.

Software developers who support DPMI can sell a single, shrink-wrapped extended DOS application that can run on multiple DOS operating environments. And users will not be required to upgrade their extended DOS applications when they switch to newer, more powerful operating environments.

The first version is 0.9; an expanded version, 1.0, should be available by the end of 1990 and will be a compatible superset of version 0.9. Products that support DPMI version 0.9 will be fully compatible with version 1.0. Reader service no. 20.
For a copy of the specification, call Intel at 800-548-4725, or write
Intel Literature Dept. JP26
3065 Bowers Ave., P.O. Box 58065
Santa Clara, CA 95051-8065

Quarterdeck announced that it is bringing its DESQview environment and API developer tools to the X Window System. DESQview/X will run X server and clients locally and simultaneously with DOS programs, as well as on a Novell or TCP/IP network. The company plans to ship the product in the fourth quarter of this year. Because the theme of Quarterdeck's third annual API developers conference in August is X Windows and DOS, X Window toolkits for DESQview will be available at that time to DESQview API developers.

DDJ spoke with Mitchell Vaughn of U.S.DATA Corporation in Richardson, Texas, who is enthusiastic about this development. His company produces software products for factory automation. Eighty percent of the computers that control the equipment and connect the factory floor with the MIS department are DOS machines, but more and more companies are investing in large systems such as Unix and VMS to run their entire systems. They therefore need a way to connect with the PCs in real time and to do it graphically. Because DEC and HP are going with the Unix strategy and are adopting X Windows and Motif as graphics standards, Vaughn "can now interoperate with DOS on DESQview and make applications that are portable. Users can view graphic displays of real-time events on DOS machines. DESQview is the ideal product. It is a true, multitasking DOS environment – a multivendor standard, unlike OS/2 and MS Windows. HP, DEC, and IBM Risc are all X Windows; with DESQview/X, I don't need OS/2 or MS Windows."

This technology brings a 3-D look and feel and iconic desktop to DESQview. DESQview/X will multitask DOS and 286 and 386 DOS extended applications, as well as local 16-bit and 32-bit X Window applications. OSF Motif and Open Look window managers for DESQview/X will be available for users running X-based workstations. DESQview/X will be compatible with EGA, VGA, EVGA, IBM 8514, and DGIS graphic display standards. This will provide users and developers with an open, vendor- and hardware-independent platform that can connect and communicate with disparate hardware from multiple vendors. Reader service no. 21.
Quarterdeck Office Systems
150 Pico Blvd.
Santa Monica, CA 90405
213-392-9851

Raima has announced the PowerCell Spreadsheet Library for professional C developers. It contains linkable functions that allow developers to add spreadsheet functionality to applications. PowerCell source code availability makes it customizable — an application can appear as an enhanced spreadsheet or as a menu-driven application with a spreadsheet component. Other C-compatible libraries can be linked with PowerCell.

Hal Kenyon of Artex Tool Corporation told *DDJ* of the benefits of PowerCell. "We have an interactive system that receives information through a variety of means, including bar coding. Anytime we query the database to get information we get an immediate feedback of the current condition — and can dump it after use because the next time the information will be updated automatically." Kenyon said that PowerCell allows his workers to import data from a database and load it into a spreadsheet, which avoids keying the information in. He also emphasized that

(continued on page 190)

(continued from page 185)
you have to be a programmer to use PowerCell — it is written almost entirely in C.

C programmers can write custom @functions, menu items, macro commands, and keystroke processing routines all in C. The WKS library is also included, with full source code. Supports Microsoft C 5.1 or 6.0 (required for installation) and MASM 5.0 or later for source code recompiles. PowerCell object code sells for $695, source code for $1495. Reader service no. 23.
Raima Corporation
3245 146th Place SE
Bellevue, WA 98007
206-747-5570

A linkable library of routines that implement interprogram multitasking, with no changes to DOS or to the compiler being used, will be available this month from **Tosh Systems** of Minneapolis. The Multi-Threading Program Toolkit will enable programmers to create self-contained DOS-based programs that multitask. Tasks can be suspended and resumed at any time or put to sleep until an event such as a keystroke or until the receipt of an intertask message or system semaphore.

Routines accompany a preemptive, time-slicing scheduler to create and remove tasks, change priorities and runbursts, send and receive intertask messages, and acquire and release system semaphores. The toolkit was handcrafted and optimized in assembly language, so it's small and fast. The system supports Turbo C, Microsoft C, and most other popular C compilers, with support for Turbo Pascal and MS QuickPascal and Basic scheduled for later release. The toolkit retails for $119.95 and comes with a 30-day money-back guarantee. Reader service no. 24.
Tosh Systems, Inc.
2627 Taylor St. NE
Minneapolis, MN 55418-2941
800-422-8674
612-788-9433

For those looking for a new approach to learning (or teaching) the C language, **The Waite Group Press** has recently released Master C, an on-line teaching system based on *The Waite Group's New C Primer Plus* (Howard W. Sams and Company, Indianapolis, Indiana, 1990). Master C presents a lesson and asks questions to assess the reader's understanding of the material. A given question may be posed as either multiple choice, true/false, or require text input, so that the question is never asked in the same manner more than once. A "Recall" mode directs the reader to problem areas. Additionally, a C glossary can be queried making the product a valuable reference for novice and experienced programmers alike.

Master C focuses on ANSI C and covers the history of the C language as well as elements of the language. Other features include the ability to monitor and retain student progress, set bookmarks within lessons, provide feedback on wrong answers, and a "slick" user interface. Master C requires 384K RAM, DOS 3.0 or later, 2.2 Mbytes of hard disk space, and a monochrome or color adapter, and can be purchased for $44.95. Reader Service no. 26.
Waite Group Press
100 Shoreline Hwy., Ste. A-285
Mill Valley, CA 94941
415-331-0575

Free EMS Toolkits are available for C developers from **Intel's Personal Computer Enhancement Operation (PCEO)**. Designed to make it easier to create expanded memory applications in C, Intel hopes these kits make the developer's job easier and the development process faster. The EMS Toolkit for C Developers consists of a set of functions for managing expanded memory in the same manner that conventional memory is managed in C. It promises to alleviate problems with page frames, 16-Kbyte boundaries, and interfacing to an assembly language driver. Call 800-538-3373 for your free copy. Reader service no. 25.
Intel PCEO
CO3-7
5200 NE Elam Young Pkwy.
Hillsboro, OR 97124-6497
503-629-7354

A user-interface class library that supports Borland's Turbo C++ has been announced by **Zinc Software**. The Zinc Interface Library (ZIL) also supports AT&T's C++ Version 2.0. ZIL allows developers to create applications that run in both graphics and text modes from one set of source code. C++ features include virtual functions, class inheritance, operator overloading, and multiple inheritance.

ZIL's event manager and window manager classes let you create flexible programs. Because the library was designed specifically for C++, you won't inherit problems associated with a "layered" implementation of older C libraries.

The event manager class is the control point for input information and message passing within a program; its two major components are an event queue and a list of devices. The event queue is a block of input elements in a linked list, and the device list contains devices polled by the event manager or interrupt devices that feed directly into the event queue.

The ZIL window manager class contains 17 robust window objects. You can build applications that allow users to mark, copy, cut, and paste between window objects, and each window object editor has an undo/redo capability and is fully customizable.

ZIL includes complete help and error systems for enhancing user interaction with a system. ZIL requires 640K of RAM, a hard disk, and MS-DOS 2.1 or later, but the company recommends MS-DOS 3.1 and a Microsoft mouse. List price is $199.95; no royalty fees are required. Reader service no. 27.
Zinc Software Inc.
405 South 100 East, Ste. 201
Pleasant Grove, UT 84062
801-785-8900

ObjectVision Inc. has come up with a way to develop object-oriented programs visually. With ObjectVision Release 1.0, you can "draw" a program's objects, interface, and database connections and then run the application in ObjectVision or convert it to C++ or Turbo Pascal 5.5. You can create new objects with a click of the mouse and then graphically add attributes and procedures to them.

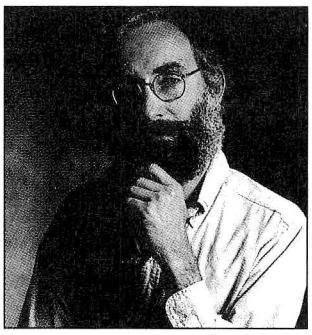
Draw a line between objects to establish a relationship; create working buttons and switches with the built-in bitmap editor; and visualize and edit object hierarchies with "3-D View." A built-in browser tracks down objects hidden in parts of the diagram you cannot see. A language for writing methods is converted automatically to code in the target language.

DDJ spoke with Andre Maziarzewski, an engineer with ABB Lummus Crest Inc. He said that "ObjectVision's powerful, practical, and flexible graphical development enviroment gives me an entirely new way of making software development faster and more productive. And because ObjectVision also reduces the effort required to learn and implement OOP, I find it an excellent tool for training in object-oriented methodologies."

Interface functions include a pixel editor, drawing functions, four font sizes, 16 drawing colors, and send to back, move to front, and align to grid functions. You can purchase a demo disk for $30 and ObjectVision 1.0 for $399. Reader service no. 28.
ObjectVision, Inc.
2124 Kittredge St., Ste. 118
Berkeley, CA 94704
415-540-4889

DDJ

The Integrity of the Product



When I was 12 years old, I spent my free time over the course of several weeks creating, with pencil and paper, parodies of *Life*, *Look*, and the *Saturday Evening Post*. I drew the pictures, wrote the copy, and invented pseudonyms for myself to put in the masthead. After 30 years these early examples of my writing no longer exist, but they were, to the best of my recollection, hilarious.

This anecdote illustrates three points.

1. My writing improves with age and forgetfulness.
2. I had a thing for magazines at an early age.
3. *Life*, *Look*, and the *Saturday Evening Post*, three magazines that served basically the same market, were sufficiently distinctive that they could be parodied by a 12-year old. This point is important, I think.

What makes something parodiable? Many attributes can be parodied: style, voice, appearance, attitude. Perhaps the one requirement is the recurrence of some constellation of attributes sufficiently cohesive as to be recognized. It's not possible to parody something that only happens once, but it's also not possible to parody something that merely recurs without generating a sense of recognition. A magazine is a package of published material that comes in the mail regularly; so are the offerings of a book club. But a magazine can be parodied and the offerings of a book club can't, because this stream of books by different authors doesn't have any recurrent aspect worthy of parody. (Some book clubs tell you about upcoming selections in a publication that has some features of a magazine; I'm not talking about that, or about a distinctive style in the company's promotional materials.)

This parodiable quality is also what made a 12-year old so fascinated by magazines. As a child, I accounted for my fascination by saying that magazines had personalities. This is an anthropomorphism that I haven't grown out of in 30 years, but while I still think that at least some magazines do have something that is well described as a personality, I no longer think that personality is the trait that makes a magazine parodiable or worthy of a 12-year old's fascination. That trait, I believe, is integrity.

The sense in which I'm using the word integrity has to do with parts fitting into a meaningful whole, systems following internal logic the premises of which are graspable from the outside, people and companies and products being true to their natures. Integrity is what makes people and magazines parodiable, it's what makes magazines work, and it's one of the things that distinguishes successful products, product lines, and companies.

The Macintosh is a product with integrity. Once you understand the desktop metaphor, you don't need any help in using the Mac. When you come up against something you haven't seen before, you feel confident that your intuition, based on your understanding of the metaphor, will tell you what to do, and the Mac usually doesn't disappoint that expectation.

Apple is not an example of a company with integrity; the continual reorganizations and the current push to produce low-margin, high-volume machines make it hard to know the company. I am certainly not suggesting that low-cost Macs are a bad idea, but I wouldn't want to be working at Apple these days.

Integrity is not just name recognition, although it encompasses that (Exxon has name recognition).

Integrity is not just consistency. Any system can be described consistently. London drivers drive on the left side of the street and San Franciscans on the right, but one consistent rule applies to driving in both cities: Drive on the legal side of the street. This even covers the tricky one-way street situations in one consistent rule that always keeps you on the right side of the street and the law. "Legal," however, is a reference to a technical manual that none carries with them while driving, while "right" and "left" are references to the driver's own body.

Integrity in a product lets the designer just throw out a few points with the assurance that the user will connect the dots. If you do it right, you don't have to do as much, because the user will bring a lot to the meeting. Interapplication communication is very nice, but it's worth remembering that the most powerful piece of software in existence lies on the other side of the screen, waiting to work with your application.

Michael Swaine

Michael Swaine
editor-at-large

15 YEARS OF POWER PROGRAMMING

NEW! PROGRAMMER'S BOOKSHELF COLUMN

#168 SEPTEMBER 1990 $3.50 ($4.50 CANADA)

Dr. Dobb's JOURNAL

SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMM

MAKING THE MOVE WITH STRUCTURED LANGUAGES

- TEAM PROGRAMMING
- PORTING TO PCs
- MOVING TO OS/2

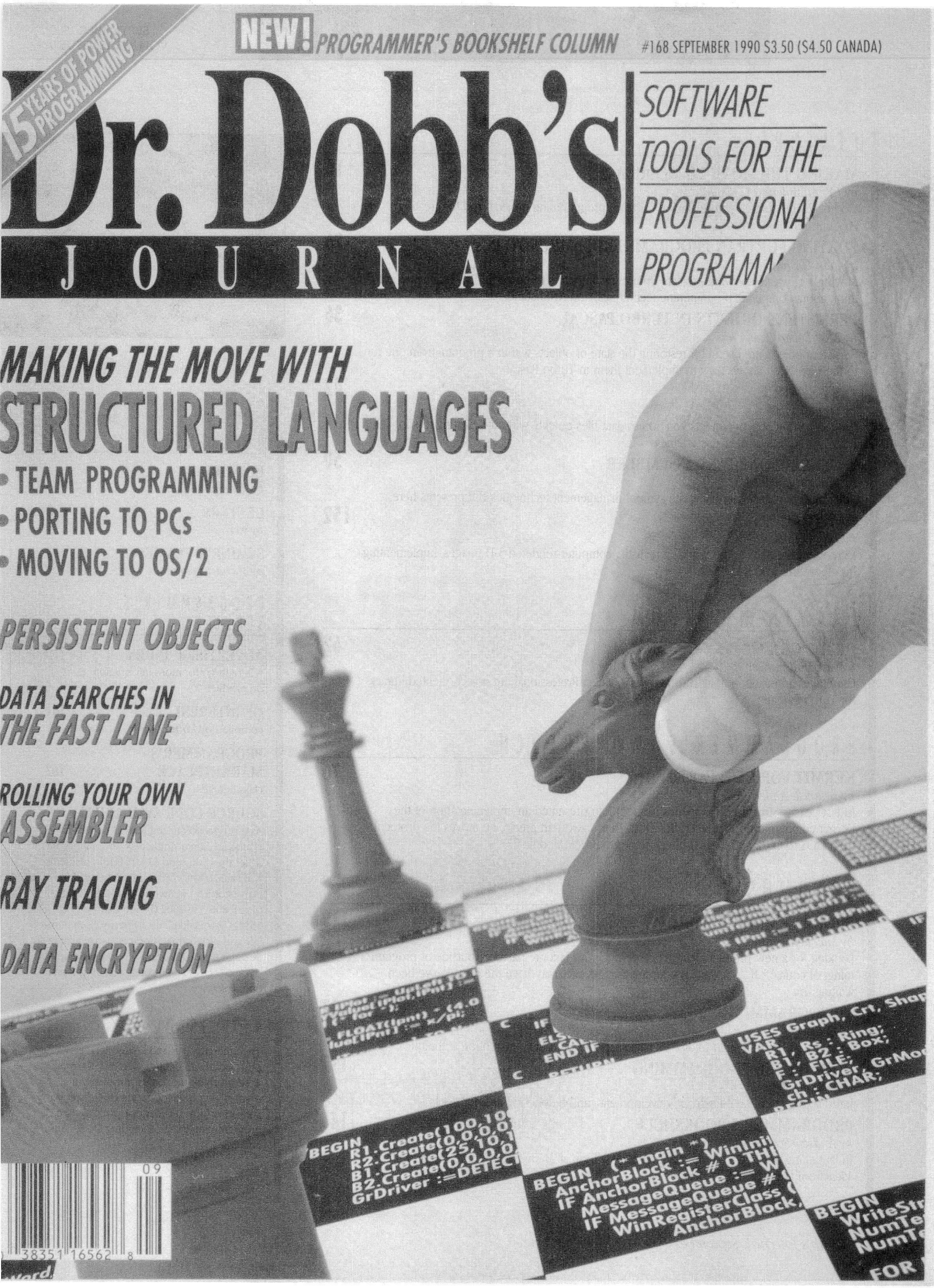
PERSISTENT OBJECTS

DATA SEARCHES IN THE FAST LANE

ROLLING YOUR OWN ASSEMBLER

RAY TRACING

DATA ENCRYPTION



09

38351 16562 8

CONTENTS

SEPTEMBER 1990
VOLUME 15, ISSUE 9

FEATURES

MAKING THE MOVE TO MODULA-2 16
by J.V. Auping and J.C. Johnston
Modula-2's modular structure is ideal for team programming projects and for creating efficient, reusable code.

PORTING FORTRAN PROGRAMS FROM MINIS TO PCs 26
by John L. Bradberry
Moving large-scale software projects from minicomputers to powerful PCs requires you to think about compilers, programming techniques, and ANSI standards.

PERSISTENT OBJECTS IN TURBO PASCAL 36
by Scott Robert Ladd
Persistent objects are useful for restoring the state of objects within a program from one run to the next; Scott shows how to implement them in Turbo Pascal.

FAST SEARCH 42
by Leon Campise
Leon's FASTSRCH program lets you access data files quickly without resorting to sophisticated linked lists or database engines.

A GENERIC ONE-PASS ASSEMBLER 50
by William E. Ives
Roll your own assembler using the symbol-management techniques Bill presents here.

RAY TRACING 152
by Daniel Lyke
Dan opens the door to the world of realistic, computer-rendered 3-D images, implementing his algorithms in C and C++.

EXAMINING ROOM

INSIDE OBJECT PROFESSIONAL 62
by Gary Entsminger
Gary takes an inside look at Turbo Power's Object Professional, an object-oriented library for Turbo Pascal.

PROGRAMMER'S WORKBENCH

KERMIT FOR OS/2: PART I 70
by Brian R. Anderson
In last year's "Structured Languages" issue, Brian presented an implementation of the Kermit communications protocol for DOS. In this two-part article, he moves the protocol from DOS to OS/2 and from Logitech to Stony Brook Modula-2.

COLUMNS

PROGRAMMING PARADIGMS 119
by Michael Swaine
Looking for a new product niche? Is gardening your favorite pastime (outside of programming, of course)? If "yes, yes" is your answer, Michael may have the idea you've been waiting for.

C PROGRAMMING 127
by Al Stevens
Al comes in from the cold to discuss hacks, spooks, and data encryption techniques.

STRUCTURED PROGRAMMING 137
by Jeff Duntemann
Jeff discovers that multi-language development can be done in bits and pieces.

PROGRAMMER'S BOOKSHELF 145
by Andrew Schulman
In launching this new column, Andrew takes a look at *Microprocessors: A Programmer's View* from a programmer's perspective.



DEPARTMENTS

EDITORIAL 6
by Jonathan Erickson

LETTERS 8
by you

SWAINE'S FLAMES 168
by Michael Swaine

PROGRAMMER'S SERVICES

ADVERTISER INDEX 160
where to go for more information on products

OF INTEREST 161
compiled *by Janna Custer*

PROGRAMMER'S MARKETPLACE 162
classified ads

SOURCE CODE AVAILABILITY
As a service to our readers, all source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (inside Calif.) or 800-533-4372 (outside Calif.). Specify issue number and disk format. Code is also available through the DDJ Forum on CompuServe (type *GO DDJ*) and through M&T's Telepath online service.

NEXT ISSUE

The emergence of DOS extenders has pumped new life into DOS, and in October we'll give you the tools for rolling your own 80386 protected-mode DOS Extender. We'll also look at other operating system programming issues, including DOS, Unix, and OS/2.

Taking Care of Business

It's always fun to bring something new to *DDJ*. That's why this month I'm particularly happy about launching the "Programmer's Bookshelf," a new monthly column devoted to books that are important for programmers. The column, which you'll find on page 145 of this issue, is co-written on alternate months by Andrew Schulman and Ray Duncan. Andrew kicks off with a look at a recently published book on the new generation of microprocessors. Unlike most CPU-oriented books that are aimed at hardware design engineers, this one is written from the programmer's perspective. As Andrew points out, you get source code instead of pin-outs.

Over the coming months, Andrew and Ray will examine books they think should be on every programmer's bookshelf. The ground rules are that books are as important to every programmer's toolkit as software and that a book should be used as a tool. They're reserving the right, however, to break the rules at any time.

If you've run across a book (new or old) you think they should take a look at, drop us a note in care of *DDJ* and we'll see if they agree.

Summer Visitors, Neural Nets, and Biocomputing

We look forward to seeing out-of-town visitors when they get a chance to drop by the office. One recent caller was Stephan Lugert of Open Network (developers of, among other tools, a nifty file comparison program called "Delta" — there's the plug Stephan) who was on his way to Seattle, taking the train up from San Diego where he'd just attended the International Joint Conference on Neural Networks.

One conference presentation that really impressed Stephan was a videotape of a research project whereby neurons were being grown on silicon plates. Nothing really new there. What was new, however, was that electrical signals generated by neural network circuits were stimulating the cells to grow in certain directions. As the dendrites grew, the cells generated growth energy and seemed to make decisions about which specific way to grow. Eventually one cell would take off while others were absorbed back into the dendrites. Intracell competition was genuinely occurring even though the growth of the dendrites appeared to be random.

There were a couple of things about the presentation that really knocked his socks off, Stephan said. One thing was the non-verbal nature of the presentation — *seeing it actually happen* instead of simply having it described. The other was that the neurons weren't static; the growth was much more dynamic and much more sensitive to the environment than researchers previously thought. And they were keeping the nerves alive for up to three months.

I'm not sure what the long range potential of research like this is, and I'd like someone to explain it to me in more depth. If you've heard about this or similar projects, I'd like to hear from you.

I don't know if we'll get a chance to look at projects like this in next April's "Biocomputing" issue (where we'll be examining neural networks, genetic algorithms, and the like), but if you have something in mind, that's the perfect forum to present what you've learned.

Inquiring Minds Want to Know

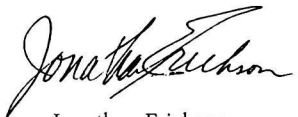
We've had a few calls asking about the art in the our July "Graphics Programming" issue. Yes, Doc Livingston of Rix Softworks created *both* the *Dr. Dobb's* blimp scene on the cover and the VGA street sign that accompanied Chris Howard's article. And yes, Doc did use his own software to generate the screens.

DDJ Online

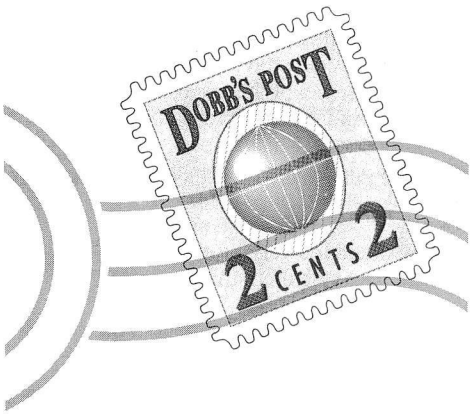
Telepath, M&T's online service, was officially up and running as of our August issue. Not only does the service give you access to *DDJ*, but it also provides you an electronic doorway to our sister publications: *DBMS*, *LAN Technology*, and *Personal Workstation*.

In addition to furnishing you with yet another way to get *DDJ*'s (and the other magazine's) source code listings, you'll find a variety of active online technical conferences already underway: 32-bit programming, C, graphics, OOPS, and discussions about dBase, Oracle, servers, Netware, and so on.

You can get to Telepath via the Tymnet network; dial 800-336-0149 to find out your local Tymnet access phone number. The communications parameters are 7 data bits, even parity, and 1 stop bit. When Tymnet answers, press *a* and, when prompted to log in, type *telepath*. You'll then be led through the sign-on process and you can go on-line immediately.



Jonathan Erickson
editor-in-chief



Rhealstone Recommendations

Dear *DDJ*,

This letter is in response to the article "Implementing the Rhealstone Real-Time Benchmark," by Rabindra P. Kar, which appeared in the April 1990 issue of *Dr. Dobb's Journal*. There are several areas of the benchmark that could be improved, which we discuss below:

1. Synthetic benchmarks and kernels are generally held in less regard now than in the past by experts in the field of performance evaluation. The detailed reasons are too long to include here. However, for an excellent explanation please read Chapter 2, "Performance and Cost," of the book *Computer Architecture – A Quantitative Approach*, by Hennessey and Patterson (published by Morgan Kaufmann, 1990). Their recommendation for evaluating performance is to use a workload consisting of a mixture of real programs. Hennessey and Patterson say that synthetic benchmarks, such as Whetstone and Dhrystone, are the least accurate of all predictors of performance. We would hope that Rhealstone can avoid some of the limitations of such benchmarks.

2. Rhealstone uses average times rather than worst-case times. In many real-time applications, worst-case time is more important than average-case. In fact, a prime characteristic that distinguishes real-time systems from data processing systems is that a real-time system has deadlines that must be met in order for the system to work properly. Average throughput is of some interest but not the critical factor. In order to determine whether the deadlines will be met, the worst-case response times must be measured.

3. The article states that the set of six time values will typically be in the tens of microseconds to milliseconds range. Note that this is a range of two orders of magnitude. However, to compute the "real-time figure of merit" the equation takes all six values and finds the arithmetic mean of them. The result is that those components whose values are small, such as interrupt latency, will contribute negligibly to the total. Similarly, large values (such as infinity for deadlock-break time on many kernels) will contribute enormously, to the point that the number of Rhealstones per second is zero. The article does allow an application-specific measurement to assign unequal weights to the components, but for the standard measurement, the weights are all equal. Hennessey and Patterson present arguments for using geometric means, rather than arithmetic means, to avoid some of these problems.

4. Regarding the definition and benchmark for measure interrupt latency, there are really three aspects of kernels that affect interrupt response: the interrupt disable time, interrupt latency, and interrupt servicing.

The interrupt disable time is the worst-case time in the kernel that interrupts are disabled (at task level), in order for the kernel to manipulate critical data structures. The processor cannot respond to interrupts at all during this time. Unfortunately, it is extremely difficult to measure this value, since there is little or no external indication from the processor that interrupts are disabled. You can measure it statistically using sophisticated hardware (emulators, oscilloscopes, and so on), but this measurement may not yield the true worst-case time. The only real way to determine it is by identifying the instruction stream in the kernel that disables interrupts for the longest period, and then calculating the timings of those instructions using worst-case timing figures for the processor. This may include cache misses, pre-fetch queue fills, memory wait states, etc.

Interrupt latency is the time it takes from the point an interrupt is acknowledged by the processor, up to the first application instruction in the interrupt service routine. The article describes one way to measure this value. For some kernels, this value is simply the time taken by the processor to respond to the interrupt, since they require no preamble or vectoring code through the kernel, but instead let the interrupt vector point directly to the interrupt service routine.

Interrupt service time is the time it takes, at interrupt level, to do whatever work is required by the kernel and application before returning to task level. Since nested interrupts at the same priority generally are not allowed during this period (although in some systems they can be), this timing can significantly affect interrupt response to bursts. This timing can be measured, given the appropriate test case. It should be recognized that different kernels have different mechanisms for servicing interrupts, and some have more alternatives than others. We suppose a particular kernel should be able to put forward its best case for this measurement.

Any benchmark attempting to define interrupt response characteristics should address all three of these factors. Rhealstone currently measures only interrupt latency.

5. The Rhealstone benchmark as described measures a small set of cases that actually measure more or less the same thing: The time it takes to switch from one task to the other in a system with 2 or 3 tasks. This approach fails to address an important aspect of kernel performance which relates to the behavior (of the kernel) when large numbers of tasks are present. Some kernel designs use linear linked-lists for handling ready tasks. This approach can be fast in the case of small numbers of tasks, but degrades when large numbers of tasks are present and are made ready in a particular order. Other designs have flat response time relative to the total number of tasks. Any benchmark should address this issue, for example by measuring task switch time in a system with a hundred tasks.

6. The standard definition of deadlock is a situation in which each member of a set of tasks is waiting for a resource that is owned by another task in the set, in such a way that none of the tasks (regardless of priority) are able to proceed. What the article refers to as a "deadlock" is called priority inversion in standard terminology.

7. Since many real-time operating systems do not implement priority-inheritance or priority-ceiling protocols, which automatically solve the priority inversion problem, the "deadlock-break time" will be infinity for such systems. Although it is legitimate to ask whether or not a kernel has a particular feature, it should be expressed as such, rather than masking it as a "performance measurement."

There are other techniques available for avoiding priority inversion in these systems through explicit control of the application. The simplest way, if the application allows for it, is to avoid having unequal priority tasks compete for the same resources. If this is not possible, the following method can be used: Let's assume the classic example of three tasks H, M, and L, with respective priorities high, medium, and low. Tasks H and L share a common re-

(continued on page 12)

(continued from page 8)
source. Task M is unrelated to H or L and does not compete for the resource. When L wants to acquire a resource, it should explicitly raise its priority to that of the highest-priority task that also uses that resource (H), before taking the resource. This requires a few extra system calls, but achieves the purpose. It behaves a little differently than the iRMX/iRMK style, which lets M run briefly until H waits on the resource. However, the former behavior may be more efficient in that respect, as it avoids two context switches into and out of M. It does block M temporarily, even if H doesn't really want the resource at all during the time L has it.

8. Rhealstone does not include a benchmark which measures "broadcast" wake-up of several tasks, using event flags. Since this feature is commonly used in real-time systems, we suggest that this be added.

9. Since Rhealstone is described in English, each implementation of it may be written in a different programming language, make different operating system calls, and make various assumptions about what the English text really means. This will make it impossible to interpret the benchmarks on different systems. To have validity, the benchmark must be written in one programming language, call one set of operating system services, and there must be one master implementation of it which everyone uses.

To summarize, we feel that these issues should be addressed to improve the validity of the proposed Rhealstone benchmark. However, even with these corrections, the accuracy in performance prediction of a synthetic benchmark, such as Rhealstone, can never come close to that of an actual real-time program. We feel that benchmarks for real-time should describe or incorporate actual real-time programs. These programs will presumably call some proprietary real-time operating system services, so the operating system calls should be converted to a standard, portable, operating system interface such as POSIX, with its Real-Time Extensions. Only then can meaningful performance measurements be obtained.

Glenn Kasten, Ready Systems
David Howard, Ready Systems
Bob Walsh, Ready Systems

Robin responds: The authors of this letter raise some interesting and valid issues about the Rhealstone benchmark. Responding to each issue in-depth would make this almost a full-length article, so I will address some of the more important objections that they have raised.

One major criticism of Rhealstones is that it specifies the measurement of average times rather than worst-case time. This criticism stems from the notion that real-time systems (hardware + real-time OS or kernel) are benchmarked primarily to validate their critical response-time capability. The expectation here is that the Rhealstone number achieved by the system should indicate if it will meet interrupt response or other deadlines with 100 percent certainty. The reality is that benchmarks are used, by the computer industry, to evaluate and compare average, long-run performance of "typical system operations," after determining that the system can meet the job's minimum requirements. *There is little doubt that any widely used real-time benchmark will be put to similar use. Consequently, Rhealstones measure average performance of "typical real-time operations." Running Rhealstones will not lift the burden of determining worst-case response times from the application designer's shoulders; it was never intended for that purpose.*

Section 4 in the letter states that interrupt disable time, interrupt latency, and interrupt servicing together affect interrupt response; and further claims that Rhealstones only measures latency. My article defines interrupt latency as the delay between the CPU's receipt of an interrupt request *and the execution of the first application-specific instruction. Their letter defines latency from the point that an interrupt is* acknowledged *by the CPU. In effect, Rhealstones measure both interrupt disable time as well as "latency," as defined in the letter. It does not measure interrupt service time, because that is entirely a function of the application, NOT the system.*

Section 5 points out that real-time system performance may be impacted if the system is running a large number of concurrent tasks. It suggests that the benchmark should measure task-switch time, for example, with a hundred active tasks. Their point is well taken but, I believe, it is inappropriate for the benchmark to specify what the "background load" should be. Why a hundred tasks? Why not five tasks or five hundred? And what should each task be doing? Background loading is a very application-specific issue. The only way to obtain a generic benchmark number is to use an unloaded system (no background application tasks).

The most important objection raised in this letter is that we should not be trying to devise or use synthetic benchmarks like Rhealstones (or Whetstones or Dhrystones) at all. A book on computer architectures is cited to support the assertion that "a mixture of real programs" can be a better vehicle for performance evaluation than a synthetic benchmark. We could have a long and vigorous debate on that point. But even if it were true, Kasten, Howard, and Walsh seem to have overlooked some major drawbacks of using a suite of actual real-time programs as a benchmark.

- *For a benchmark to gain wide acceptance in an industry, it must be compact and easy to recompile and run in a variety of software/hardware environments. This is much more true of a synthetic benchmark than an actual program suite. It is probably the major reason why Dhrystones and Whetstones are so commonly used, whatever their theoretical shortcomings.*
- *Since synthetic benchmarks are short and relatively simple, the application engineer can easily understand them, modify them and/or decide if they are appropriate for the task at hand. In the real world, few engineers have the time to even understand what exactly is being measured by some humongous program suite. If you were evaluating a real-time kernel for use in an automobile microcontroller, how much confidence would you have in a mixture of benchmark programs from the oil-drilling industry, satellite manufacturers, nuclear reactor designers, and a dozen other industries (especially if you did not know what any program actually did)?*
- *The program suite would not reveal what the system's worst-case response times are. This was one of their major criticisms of Rhealstones, but their suggested solution comes no closer to addressing it.*

Finally I'd like to thank Kasten, Howard, and Walsh for their thoughtful response to my benchmarking proposal. The absence of a widely used real-time benchmark standard often leads to the use of inappropriate general-purpose benchmarks (Whetstone, Dhrystone) by real-time application designers. The Rhealstone benchmark was proposed to help fill this void, though neither Rhealstones nor any other benchmark will do justice to every real-time situation. If it seeds and motivates further creative effort and discussion in the real-time software community, the Rhealstone proposal will have more than served its purpose.

(continued on page 14)

(continued from page 12)

SEGTABLE and Windows 3.0

Dear *DDJ*,

Facing the start of a Windows development, I recently reread the article which Tim Paterson and Steve Flenniken contributed to the March 1990 issue of *DDJ*, "Managing Multiple Data Segments Under Microsoft Windows." It was a good article. But I have a few questions about its applicability to Windows 3.0.

The obvious first question is does Windows 3.0 support the undocumented call they use to register the local segment table? Since recent Microsoft apps that ran under Win 2.0 and that they say use these techniques run unchanged under Win 3.0, I would guess that the answer is "Yes, the world does look a bit different, though, in Win 3.0 since the segment table is no longer handling segment numbers but rather protected-mode segment descriptors." Again, I would guess that this shouldn't make too much difference.

In the second article in the series, Paterson and Flenniken spend time discussing how the segment table technique fits in the EMS. Since Win 3.0 manages all memory on a machine, I would think that one need no longer worry about EMS but should write an application as if one had much more global memory available for allocation. One might still be memory constrained on a machine with 1 Meg of RAM running in Win 3.0 real mode, but now it seems that the solutions are to write to use less RAM (which might include roll-your-own virtual memory) or to require more RAM installed as extended memory and use Win 3.0's protected memory mode.

I actually find it a bit disturbing that Microsoft uses these techniques in their products but does not document them. It seems to give the lie to their assertion that Microsoft apps writers have no secret information about Windows that gives them a competitive advantage.

In addition to commenting on the above, I would appreciate any other comments Tim Paterson might have with regard to using memory in Win 3.0.

Steve Williams
3Com Corporation
Santa Clara, California

Tim responds: Steve asked some very good questions about the relationship between Windows 3.0 and the segment table I described in my February/March article. Fortunately, the answers are surprisingly simple, once you know a little bit about the new Windows.

Windows 3.0 has three operating modes, selected when it starts up. Real mode is identical to Windows 2.x, so it can run all Windows 2.x applications unchanged. Real mode even works on 8088/8086 processors, just as Windows 2.x did. Real mode also supports the segment table exactly as I described in my article. In fact, the new Windows Software Development Kit (SDK) includes a description of the DefineHandleTable() *function that kicks in the segment table, although the one-paragraph description is a little thin to relate a full understanding of its application.*

The other modes for Windows 3.0 are Standard mode (or 286 mode) and 386 Enhanced mode. From the standpoint of a Windows application, these modes are the same; their main difference is in how they handle non-Windows programs. These modes run the processor in protected mode, where the values loaded into the segment registers are "selectors," not paragraph addresses. Even when the Windows memory manager moves or discards a segment in protected mode, the selector never changes; as far as the application is concerned, there is no memory movement. The concept of the segment table is meaningless, and the DefineHandleTable() *function is ignored.*

In other words, if you're willing to limit your applications to 286 or better processors running Windows 3.0, ignore my article. Not only will you get to pass far *pointers around willy-nilly with no concern for memory movement, you will also get transparent access to extended memory. But if you want to run on the older machines (8088/8086) or with Windows 2.x, Windows 3.0 Real mode changes nothing. It appears that most companies, including Microsoft, will be taking the first approach.*

I, too, found it disturbing that Microsoft applications were using a technique that was not documented — that's why Steve Flenniken and I wrote the article. However, people at Microsoft left the impression that this was more of an oversight (or maybe just too much trouble to document) than intentionally holding back. I find myself working for Microsoft once again, and no one has complained to me that I told a secret.

Hypertext Caveats

Dear *DDJ*,

Regarding your June 1990 hypertext issue: How do I usually read *DDJ*? I usually read it on the train or as bedside reading. I also usually mark up the listings. Hypertext has its place, but not in the quiet contemplation that must accompany learning.

John O. Goyo
Port Credit, Ontario, Canada

Patents Cont.

Dear *DDJ*,

Lacking Barr Bauer's special qualifications and insights into the patent ethos, I would nonetheless like to reply to his letter which appeared in the July 1990 edition of *DDJ*. Mr. Bauer makes a convincing case that an inventor or investor should be encouraged to advance the cause of technological innovation. Whether patent laws do this or not is not clear.

It appears that patent laws can benefit large, well-financed organizations, but this benefit comes more from the ability to defend themselves in court than from any innate protection afforded by the laws themselves. Many innovations of the twentieth century (button-release socket wrenches, intermittent-control windshield wipers, FM radio, television, to name only a few) have been appropriated from their original inventors and exploited by large organizations. And, ironically, a significant part of the success story of American industry in the late 1800s-early-1900s and beyond is the story of industrial espionage and the infringement of some key European patents.

Whether or not investors and inventors benefit in the long run from patent laws, it seems to me that a larger question when the laws are applied to software is whether the craft or society itself benefits. As Mr. Bauer points out, changes in software come so rapidly that patents may outlive their usefulness well before they expire. If this is so, how can developers ever hope to build on existing software to further the state of the art?

I think patenting will tend to further fragment the development of software, adding unnecessary costs and delaying true innovation. Isaac Newton said "I have stood on the shoulders of giants," and by this he meant: No programmer is ever going to get anywhere if he has to go back to square one every time he boots his system.

Phil Wettersten
Chillicothe, Ohio

We welcome your comments (and suggestions). Mail your letters (include disk if your letter is lengthy or contains code) to DDJ, *501 Galveston Dr., Redwood City, CA 94063, or send them electronically to CompuServe 76704,50 or via MCI Mail, c/o* DDJ. *Please include your name, city, and state. We reserve the right to edit letters.*

DDJ

Making the Move to Modula-2

Modular structure is important for multiprogrammer projects

J.V. Auping and J.C. Johnston

Three years ago, we had occasion to upgrade a group of small scientific computers used for laboratory data acquisition. We chose to standardize all of our new software efforts and use Modula-2. In this article, we'll talk about why we chose Modula-2 and about the benefits and drawbacks of the language.

As support programmers for a materials research laboratory, we are involved in writing application programs that perform control functions and acquire data from special-purpose experimental furnaces. There are about a dozen such furnaces for controlled heating and cooling of metals and alloys and for crystal growth activities. Each furnace had been initially equipped with a data acquisition and control system made up of a small desktop computer that was programmable in Basic and an IEEE-488-based data acquisition unit. As our researchers came up with more complex experiments, we were limited by the speed and memory capacity (32K words of user space) of the desktop computers.

In addition, as the researchers became familiar with the capabilities of the PC systems appearing in their offices, they grew dissatisfied with the small display and slow tape drives of the data acquisition computers. The decision was made to replace the old computers with faster, larger, PC-type systems. We had been quite satisfied with the capabilities of the data acquisition units, and also had invested significant amounts of time in the wiring of thermocouples and other transducers. We were determined to keep them with whatever new systems we developed. This put some constraints on our choice of hardware.

Judy Auping and Chris Johnston work in the Microgravity Materials Science Laboratory at NASA's Lewis Research Center in Cleveland. They both hold Ph.D.s in chemistry from Cleveland State University. Chris Johnston's book, The Microcomputer Builder's Bible, *was published by Tab Books in 1983. They can be reached at 216-433-5016 and 216-433-5029, respectively.*

After a survey of the available computers, we settled on an 80286-based AT-compatible system with a 40-Mbyte hard disk and EGA display. We also chose an IEEE-488 interface card that promised code to support most of the popular programming languages. Since we had been dissatisfied with the cryptic nature of even our own most carefully written Basic code, we were pleased to have the chance to move to a more satisfactory programming language. This was an important decision for us, because we wanted to standardize on one language throughout the lab. As the different furnaces have many functions in common, we wanted to be able to share code.

Looking at Languages

The first language we considered was Pascal, as one of us had had considerable experience with it prior to this project. After trying to apply Pascal to our experiments, we decided we liked the strong typing and structured design of the language, but were concerned that it would be unable to support a joint programming effort. We had been intrigued by literature[1,2] that described Modula-2 as a language designed for team programming. Its similarity to Pascal was convenient, and it seemed well-suited to joint projects that required low-level access to the computer hardware. So, we decided to test a Modula-2 development system.

We were particularly interested in and wanted to evaluate a number of features described in the articles on Modula-2:

- The modular structure, which would allow two or more programmers to work in a controlled manner and permit code interaction without unforeseen side effects.
- Transparent access to the low-level facilities of the machine without disturbing the rest of the program.
- Increase in program reliability because of built-in type checking.
- The idea of creating independent modules that can be used in other programs.

A number of things looked as if they would be real (if minor) annoyances:

- The issue of case sensitivity. All of Modula-2's identifiers are case sensitive, and reserved words are required to be uppercase.
- Modula-2's inability to do I/O on complex types (records and arrays, for instance) directly, without byte counting.
- The relative scarcity of commercial software libraries available for Modula-2.

After installing the Modula-2 development system and writing a few short test programs, we were sufficiently encouraged to embark on a fairly major "learning project." We looked around for something that would be a good multiprogrammer test project and that would be useful to us later if we decided to standardize on Modula-2. At that time, nearly all of our experiments were already running on the smaller computers, generating data that required plotting. Our researchers had also begun clamoring for the ability to view their data graphically while an experiment was in progress. To evaluate Modula-2's ability to solve our problems we decided to implement a library of screen and plotter graphing routines for our data acquisition and control programs.

Developing Modula-2 Libraries

The first step in developing our library was to determine exactly what capabilities it would provide. We were pleased with the plotting functions available in the Basic ROMs in our small computers, and we decided to include many of these functions but improve on the calling syntax. This gave us a base set of "high-level" procedures that would have to be implemented. These would be the only procedures invoked directly by an applications programmer using the library. We defined a number of other functions that would be needed either as additions to the base set or as lower-level functions required to implement the base calls.

Once we had an idea of what we had to write, we needed to break the project into two more or less equal pieces. Based on our examination of the procedures we had already identified, the project seemed to divide naturally between the low-level hardware manipulation functions and the higher-level graphing calls. The procedure for drawing an axis doesn't need to know about how the lines are drawn or that drawing lines on the display and the plotter are handled differently. On the other hand, the line drawing procedure doesn't need to know what is being drawn. Table 1 lists the low-level hardware manipulation and high-level plotting functions. Although it may look like an unequal division of labor, most of the low-level function calls contain separate code for each plotting device included in the system; so the low-level section required more lines of code.

In Modula-2, all modules except for the main program module have two parts: A definition module that formally defines what procedures and variables are available, and an implementation module where the executable code to perform those functions resides. The definition module for the high-level plotting functions, *ModPlot*, is shown in Listing One, page 76, and the module for the low-level functions *GDriver* is shown in Listing Two, page 76. We also created the module *DataDefs* (Listing Three, page 76) to hold all of the global types. By defining color types, device types, and so forth, we are able to use statements such as *SetBackgroundColor(Blue)* or *SetPlotDevice (EGA)* rather than a cryptic numerical code. The price we pay for this increased program readability is that the relevant types must be explicitly imported from *DataDefs* into every other mod-

Dr. Dobb's JOURNAL SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

PUBLISHER *Peter Hutchinson*

EDITORIAL
EDITOR-IN-CHIEF *Jonathan Erickson*
MANAGING EDITOR *Monica E. Berg*
TECHNICAL EDITORS *Michael Floyd, Ray Valdes*
ASSOCIATE MANAGING EDITOR *Janna Custer*
CONTRIBUTING EDITORS *Al Stevens, Jeff Duntemann, Martin Tracy, David Betz, Tom Genereaux, Andrew Schulman*
COPY EDITORS *Rhoda Simmons, Pamela Dillehay, Nan Fornal*
EDITOR-AT-LARGE *Michael Swaine*

ART/PRODUCTION
ART/PRODUCTION DIRECTOR *Larry L. Clay*
ART DIRECTOR *Michael Hollister*
PRODUCTION SUPERVISOR *Amy Shulman Lesovoy*
TYPOGRAPHERS *Margaret Anderson, Charlene Carpentier*
COVER PHOTOGRAPHER *Michael Carr*

CIRCULATION
DIRECTOR OF CIRCULATION *Maureen Kaminski*
CIRCULATION MANAGER *Randy Robertson*
CIRCULATION PLANNING MANAGER *Manny Sawit*
DIRECT MARKETING MANAGER *Francesca Davies*
NEWSSTAND MANAGER *Sarah Forsman*
DIRECT MARKETING COORDINATOR *Susan Bauman*
PROMOTION COORDINATOR *Philip Tsang*
FULFILLMENT COORDINATOR *Anne Jean*

ADMINISTRATION
VICE PRESIDENT OF FINANCE *Kate Deschamps*
CONTROLLER *Mary Collopy*
CREDIT MANAGER *Betty Arsene*
ACCOUNTING SUPERVISOR *Renate Kernke*
ACCOUNTS RECEIVABLE *Wendy Ho*
ACCOUNTS PAYABLE *LuAnn Rocklewitz*

MARKETING/ADVERTISING
ASSOCIATE PUBLISHER *Karla Spormann*
ADVERTISING COORDINATOR *Laura Stack Pullen*
MARKETING ASSISTANT *Sara Noah Ruddy*
ACCOUNT MANAGERS *see page 160*

M&T PUBLISHING INC.
CHAIRMAN OF THE BOARD *Otmar Weber*
DIRECTOR *C. F. von Quadt*
PRESIDENT *Laird Foshay*
VICE PRESIDENT OF PUBLISHING *William P. Howard*

DR. DOBB'S JOURNAL (USPS 307690) is published monthly, except semimonthly in December, by M&T Publishing, Inc., 501 Galveston Dr., Redwood City, CA 94063; 415-366-3600. Second-class postage paid at Redwood City and at additional entry points.

ARTICLE SUBMISSIONS: Send manuscripts and disk (with article, listings, and letter to the editor) to the associate managing editor 415-366-3600.

***DDJ* ON COMPUSERVE:** Type GO DDJ.

SUBSCRIPTION: $29.97 for 1 year; $56.97 for 2 years. Foreign orders must be prepaid, including the additional postage (air or surface) in U.S. funds drawn on a U.S. bank. Add $13 per year for surface mail; add $36 for airmail to Canada and Mexico; or $26 for airlift to all other countries.

POSTMASTER: Send address changes to *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. **ISSN 1044-789X**

CUSTOMER SERVICE: For subscription questions, call toll-free 800-456-1215. For subscription orders or change of address call toll-free 800-456-1215 (U.S. and Canada) or write *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. For book/software orders call 800-533-4372 (in California 800-356-2002).

FOREIGN NEWSSTAND DISTRIBUTOR: Worldwide Media Service Inc., 115 E. 23rd St., New York, New York 10010; 212-420-0588 FAX 212-420-1265.

Entire contents copyright ©1990 by M&T Publishing, Inc., unless otherwise noted on specific articles. All rights reserved.

The Audit Bureau

ule using those types. Note that *DataDefs* is an example of a Modula-2 definition module with an empty implementation module. It allows us to define a number of types and variables to be imported whenever necessary but contains no executable code; its only function is to define the variables.

In our case, determining which of us would do what part of the project was easy. The division into high-level (user-related) and low-level (hardware-related) functions matched our interests fairly well. But we still had to decide how to pass information from one part to the other and how to handle range checking and error reporting. Since we already had the *DataDefs* module for defining global types, we decided to use it to also hold a set of global variables that would contain the current settings for the graphics parameters. This would reduce the number of formal arguments needed to pass information. We decided to do all of the checking for the validity of data in the high-level procedures. The procedures in *GDriver* trust that they are being passed legal pixel coordinates by the code that calls them.

Part of the idea behind Modula-2 is that it allows programmers to define an interface between modules that is stable and isolates one module from another. In theory you should think deep thoughts about the interface (the definition module), define it, and never change it again. In our experience, reality was only a little different. Both the high-level *ModPlot* definition module and the low-level *GDriver* definition module were remarkably stable once we had thought about them for a while. *GDriver* did require one addition, the *CleanUp* procedure, unforeseen in the original design.

Modula-2 has delivered on its promise to be very suitable for multiprogrammer projects

(The cleanup procedure was needed because of the way one of the compiler-supplied library modules managed the interrupt system. Another compiler might handle things differently, making the additional procedure unnecessary.) We made a few minor changes to the types of the arguments in some of the *ModPlot* procedure calls. The *DataDefs* module, however, an important behind-the-scenes part of the interface, was a lot more fluid.

The actual process of implementing our library was straightforward. We spent two weeks defining and partitioning the functions. We then went our separate ways to do the coding and some independent testing of the sections. Obviously, as in any hierarchically divided project, it was more satisfying to test the low-level modules. Testing of the high-level *ModPlot* functions was limited to producing textual output such as "Call to DrawAbs made with the following arguments . . . ", whereas the testing of the low-level procedures generated pretty pictures right from the start. On the other hand, when the low-level modules malfunctioned they were harder to debug, as the errors usually were not obvious and sometimes were catastrophic. After about a month of separate coding and testing, we were able to link the two sections together in a test program. We spent a relatively short time using test programs to debug the library. The whole project took about two months (four man months) from start to finish.

Listing Four, page 77, and Figure 1 show the source code and output of a program, *Example*, which demonstrates the use of the *ModPlot* library. Note that *Example* imports types from *DataDefs* that do not explicitly appear in variable declarations, but do have instances appearing in the code. For example, *SizeType* must be imported because *Small* is used in the statement *SetCharSize(Small)*.

Team Programming and Modula-2

Of course, the big question is, how did the use of Modula-2 affect the process of writing a team-programming project? Our feeling is that the modular nature of the language streamlined the process of writing and testing each function. Granted, this could be done with any "good" structured language, but the formal, definition module/implementation module structure made it particularly easy to isolate and group similar procedures.

Comparison with our past experience shows that the strict typing of Pascal or Modula-2 eliminates the source of a lot

Low-level Hardware Manipulation Functions

Determine what graphics devices are available
Move the active position to the specified device coordinates
Draw from the active position to the specified device coordinates
Draw a label string
Clear the video screen and set to text, graphics, or menu (no cursor) mode

High-level Plotting Functions

Initialize the graphics functions
Choose a plotting device for subsequent output
Set plotting boundaries
Set boundaries for a scaled area
Set the scale for the scaled area
Set the type of plotting units
Set the size of characters for subsequent labels
Set the angle for subsequent labels
Set the line type for subsequent draw commands
Set the pen color
Set the background color
Set the label orientation relative to the active position
Set the number of digits to the right of the decimal point
Return the ratio of the physical dimensions of the plotting area of the current device
Set the character font to be used in subsequent labels
Draw a line from the active position to the specified coordinates
Move to the specified coordinates
Do an incremental draw from the active position
Do an incremental move from the active position
Draw an X-axis with tic marks
Draw a Y-axis with tic marks
Draw a set of X-Y axes with tic marks
Draw an X-Y grid
Draw a label at the active position according to the current character size, label orientation, and label angle settings
Draw a set of X-Y axes with tic marks, labelled according to the current character size and fixed decimal settings
Draw an X-Y grid, labelled according to the current character size and fixed decimal settings
Draw a box around the current plotting area
Draw a symbol of specified size and type at the active position
Return the coordinates of the active position
Clear the screen and set the mode

Table 1: *Library functions*

(continued from page 18)

of errors that creep into Basic or Fortran programs. In comparing Pascal to Modula-2, it is more difficult to see a great difference. The tendency (in well-written programs, at least) for Modula-2 programs to be broken into many small parts makes the isolation and elimination of errors easier than in Pascal and makes Modula-2 code more reliable from the outset than Pascal or C code. We have, of course, found occasional bugs in our library during the past year and a half. Almost all of these, however, have been due either to errors in logic or to timing problems caused by moving to machines with different clock speeds.

The reusability of modules is an important part of the rationale behind Modula-2. Once you write and debug a module you can put it in your library and use it whenever you need that function again. A problem arises in modules that import information from other modules. *GDriver* imports a module called *HdwDefs*. That means that *GDriver* can't be used in another program unless *HdwDefs* is available too. We have found that the most easily reusable modules are often the lowest-level ones that perform one simple, clearly focused function. Reusability of more complicated modules must be planned for in the original design of the code because you need to minimize the number of dependencies.

On the other hand, *GDriver* is a general-purpose drawing and labeling module available for any program needing it. This approach is preferable to writing the same functions again the next time they are needed. In a broader sense, this was the purpose of the entire plotting package. It provides a simple interface for a programmer who needs to add some simple graphing functions to a program. As a byproduct of its creation we now have a number of modules that we can use anytime.

The issue of transparent access to low-level facilities was also important to us. Some of the functions that we require in our experiments involve the use of boards that plug into the AT bus. Many of these require special software to access their registers and onboard memory. Modula-2 has enabled us to do that quite successfully. Some of these devices use interface code that is wholly or partially written in assembly language but is called from a Modula program as if it were any other module. Modula-2 can access individual memory locations and I/O ports, allowing the programmer to do much of this manipulation without resorting to assembler. It is, however, a good idea to isolate any such extremely machine-dependent code in its own module.

Error handling has been and continues to be more of a problem. If an external error occurs deep in a nested set of procedures, there is no easy way of invoking some error-handling procedure unless an error flag is tested each step of the way back. For example, if the plotter goes offline during a plot (someone inadvertently turns it off or the paper-load lever is moved), the transmission code deep in the low-level routines will eventually time out, generating a serial error. The error is posted in a variable in *DataDefs*

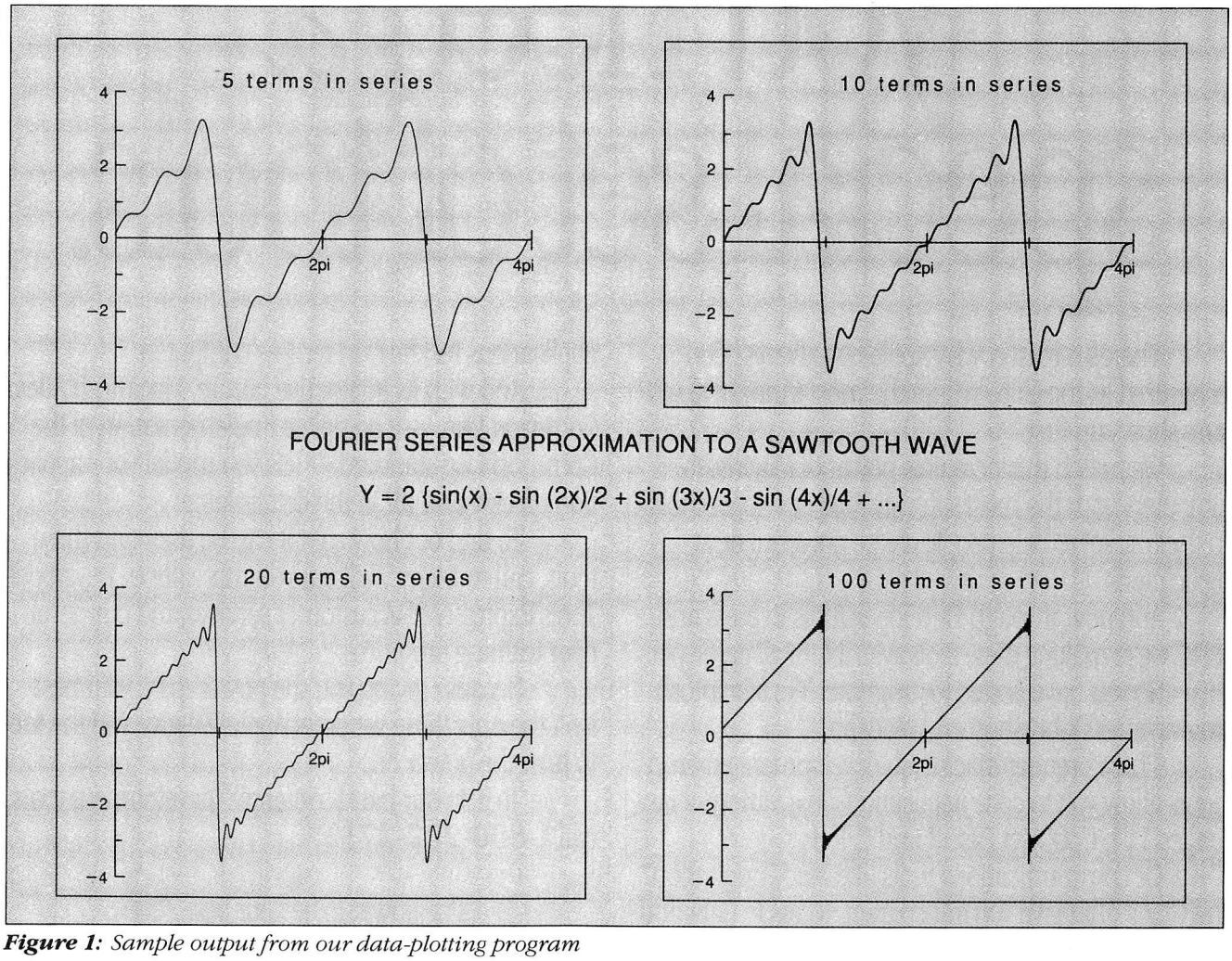


Figure 1: *Sample output from our data-plotting program*

(continued from page 20)
along with a string explaining the problem. There is no way to guarantee that the calling program will notice this condition unless the error flag is checked after every call to *GDriver*. This means that every call to *DrawAbs*, *MoveAbs*, or *DrawString* in *GDriver* has to be checked, and the rest of the calling routine short-circuited, returning an error indication to the external calling program. This issue has not been resolved to our satisfaction. Our working solution has been to have the low-level code that discovers the error post a message to a reserved area on the display screen explaining the problem and giving the user a chance to fix it.

The Upside and Downside of Modula-2

One unanticipated benefit of Modula-2 has been that it allows us to maintain our own coding styles and habits and a sense of ownership of our own code without impeding the collaboration necessary to make a project like this work. (As you may have gathered, we have not progressed to the point of "egoless" programming. We are, at best, pursuing "ego-reduced" programming.) For example, one of us likes the indented structure:

```
IF SomeCondition THEN
  DoThis;
  ThenThat;
ELSE
  DoSomethingElse;
END (* if *);
```

while the other prefers:

```
IF SomeCondition
  THEN
    DoThis;
    ThenThat;
  ELSE DoSomethingElse;
END (* if *);
```

The fact that we can each write and maintain our own code in separate modules means that neither has to put up with the other's unspeakably ugly IF-THEN style.

We expected that the language's case sensitivity would turn out to be an annoyance. Instead, we have found that once you are used to the rules it isn't a burden at all, especially if you use a syntax-assisted editor. In fact, we have both grown to like the fact that reserved words are all in caps. Using normal capitalization for variable and procedure names allows you to quickly pick them out from the reserved words in the code. Case sensitivity forces you to clean up the variable names so they all match exactly. In Pascal, case insensitivity allows you to put off cleaning up the code to make it prettier. Modula-2's case sensitivity does have a drawback in that it makes some new errors possible. It is legal to declare a local variable with the same name as a global variable. No changes can be made to the global variable inside the procedure where the local variable is declared, protecting the global from unforeseen side effects. However, if you miscapitalize the declaration of the local you can inadvertently change the global. This situation actually occurred in some code a student wrote for us, and it took quite a while for us to help him locate the problem.

A real annoyance with Modula-2 is that it has no equivalent for Pascal's *get* and *put* statements, which allow a program to automatically handle I/O on complex data structures such as records. In order to write a record variable in Modula-2, a program has to calculate the size of the record and then write that number of bytes. Presumably, this is in

(continued from page 22)
keeping with the philosophy that all I/O in Modula-2 would be through a procedure specific to the data type involved. It would be nice if there were a more automatic way to do this. The random access I/O of data to/from disk files has the same kind of limitation, also probably for the same reason. It seems that the compiler could do this for you.

Another drawback of using Modula-2 becomes evident if you are writing a short program from which you want quick results. There is some significant overhead in typing in the numerous *FROM SomeLibrary IMPORT SomeProcedure;* statements necessary to do any useful task. For a while, we both returned to Turbo Pascal when we had to do something quickly. Eventually, as we became more involved with Modula-2, we evolved template main program modules that contain the imports (such as *WriteLn, WriteString, Assign, Concat,* and so forth) that we typically use in our work. However, there are still the frustrating times when you discover at compile time that you've forgotten to import one or more of the procedures you need.

The scarcity of commercially available software libraries for Modula-2 was not an insurmountable problem for us. We invested about six months in writing our own utility routines, including both a program to produce straightforward scientific plots and a useful set of generic field editing routines. We have also successfully used packages designed to be called from other languages, with more or less trouble depending on what memory model the library uses. One image-processing library written to be called from Microsoft Pascal required an assembly language interface module to rearrange the stack so that the arguments were passed correctly.[3]

Conclusions

After extensive use, we have both become rather fanatical about Modula-2 and have standardized on it for all new development in the lab. We have hired undergraduate and graduate students for summer jobs who were able to learn the language quickly and produce modules for us in reasonably short periods of time. We have certainly met resistance from programmers unfamiliar with Modula-2, but we have adhered to our decision to maintain a single language.

In short, we feel that Modula-2 has delivered on its promise to be very suitable for multiprogrammer projects. Its modular structure has allowed us to write efficient, reusable code that can be used in a variety of applications. At this point, we're looking forward to more support, both commercial and user group.

REFERENCES

1. "Modula II, An Overview — Excerpts From A Talk By Niklaus Wirth," *Micro Cornucopia* (Aug/Sept 1985), 25.
2. N. Wirth, "History and Goals of Modula-2," *BYTE* (August 1984), 145.
3. C. Johnston, "Modula-2 in the Mainstream," *Computer Language* (June, 1989), p.71.

NASA does not endorse commercial products. Details about any products named in this article were included for completeness and accuracy. No endorsement or criticism of these products by NASA should be assumed. The source code for the entire package is available from the authors upon request.

DDJ

(Listings begin on page 76.)

Vote for your favorite feature/article.
Circle Reader Service **No. 1.**

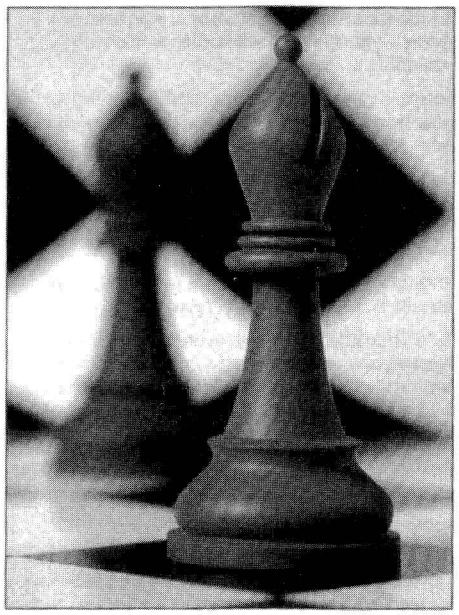
Porting Fortran Programs from Minis to PCs

A method to prevent software madness

John L. Bradberry

Improvements in personal computer hardware, operating systems, and compilers have placed PCs in a competitive role in applications traditionally dominated by mainframe computers. Developers of scientific applications such as radar and antenna software have traditionally defined "state of the art" in the digital world at all levels of electronic instrumentation. Historically, PCs were limited in clock speed, extended precision computing power, and compiler sophistication. As a result, the scientific community restricted PC uses to intelligent terminals, word processors, and spreadsheets. In only the past few years, these limitations have almost been completely resolved and PCs are rapidly becoming viable options in all levels of problem solving.

This article addresses issues related to porting large-scale software from mainframes to the PC. Issues related to compilers, programming techniques, ANSI standards, and some PC software resources will also be examined. For the purposes of example, the structure of Microplots, a Fortran-based graphics system, is used to illustrate a structured programming design approach. Microplots was ported successfully from a minicomputer to the PC using Microsoft Fortran 5.0.



John is senior research engineer with the Georgia Tech Research Institute, specializing in radar and antenna research. He is also development manager for Scientific Concepts, where he can be reached at 2359 Windy Hill Road, Suite 201-J, Marietta, Georgia 30067.

Minicomputer Operating Systems Versus PC Development Environments

For many years, the most popular minicomputer operating systems and compilers were products offered by the hardware vendors. VMS from Digital Equipment Corp. is a good example. VMS maintains widespread support as a good solution to problems involving huge arrays for 32- to 64-bit number crunching problems. Although the real-time responsiveness of VMS has often been questioned, DEC made many improvements to Fortran 77, and its customers responded with millions of lines of code dedicated to VAX Extended Fortran. Much of this resulted in code with virtually no chance of portability.

Another example is RTUX from Hewlett Packard. Over the last two years, HP has adopted a modified version of real-time Unix (RTUX). As a Unix-based operating system, RTUX offers virtual memory management and additional connectivity options. However, Fortran development is at best secondary in most Unix environments.

With DOS-based PC architectures, third-party clone vendors created a potpourri of hardware and software to steadily increase the PC's feasibility as a scientific problem solver. The single-task nature of DOS not withstanding, the increase in clock speed and development support software proved to be irresistible to the scientific community. However, full-featured minicomputer development environments still offer features not easily addressed under PC DOS. Some of these features include:

- Virtual memory access (>640K) transparent to the process.
- Full featured system services librar-

(continued from page 26)

ies such as spawning child processes from within a program shell, direct keyboard/memory operations with or without wait, and I/O such as IEEE and 8- to 16-bit Digital Control and DMA (Direct Memory Access) Channels.
- Symbolic debuggers extended to source level with access to user and system library modules.
- User-controlled global variable space with inter-process communication capability (memory based!).

It should be noted that some of these issues have been addressed in OS/2 and to a limited extent with 386/486 development utilities (DesqView for example). However, the vast majority of DOS applications and the scope of this article is limited to DOS 3.x and 4.x versions.

Converting Software from Minicomputers to PC Platforms

In an ideal world, application portability from one machine to another would simply involve moving the source code from machine to machine, then recompiling and linking all program modules. Unfortunately, only a small number of relatively simple programs ever make this a reality. The main reasons for this phenomenon are readily apparent: CPU and compiler-dependent code grows in direct proportion to the size and complexity of an application. Inexperienced programmers and program managers are often unaware or unconcerned about portability. In addition, issues of portability sometimes surface after hardware or software obsolescence becomes imminent. (See the accompanying text box for some *Do's and Don'ts* of developing portable software.)

A structured design approach can be essential as preventative maintenance for software support

In the rest of this article, I'll discuss porting a large graphics application from a mainframe computer architecture to a PC. The application, called "Microplots," is a collection of scientific plotting routines for two-dimensional and three-dimensional data files. The Microplots routines contain over 11 algorithms originally designed for VAX and other minicomputers. (For a demo disk of the Microplots system, contact the author at the address at the beginning of this article.)

The compiler chosen for porting Microplots to the PC was Microsoft's Version 5.0 of Fortran 77. It was chosen primarily because:

- System service features such as command line access, child process spawning, and programmable operating system functions were desired.
- Mixed language calls were necessary for unsupported system services utilities such as keyboard peeks (*read_without_wait*). Microsoft supplies a full compliment of mixed language support for C, Assembler, and Basic.
- Fortran calls to lower-level graphics primitives were essential for this application.
- ANSI Fortran 77 extension support.

Designing for Portability

A structured design approach can be essential as preventative maintenance for software support. The term "top-down" software design has become a cliché over the years. In reality, the top-down programming approach is usually complimented with a "bottom up" implementation. The resultant layering of software will ultimately result in the form of an inverted pyramid, as illustrated in the Figure 1.

Lower layers represent the device and CPU-specific implementations of key functions. At the device driver level (layer 2 in the figure) other third-party vendor software can be merged for completeness. The following features of this structure readily support efforts in the port process:

- As the tree is implemented, the top two levels maintain a fairly generic Fortran 77 form. All code at these levels exists in a form virtually application and CPU independent.
- Each layer provides virtually unlimited expansion of capability by the easy addition of plot algorithms or graphics devices with little impact on existing code.
- As suggested by the pyramid structure, the software most affected by changes in CPU or graphics devices represents the smallest amounts of code in the entire system. Of the approximately 100K lines of source code in Microplots, less than three percent (the bottom of the pyramid) is device or CPU dependent.

Source Code Port Example

The plot program that generates the globe shown in Figure 2 is shown in Listing One (page 80). Prior to the port, the globe program contained many poor programming practices that made the port process cumbersome. For example, most variables were not typed (IM-

Top Level: Microplots Graphics Application Software
Contour Plot - Polar Plot - Linear Plot ... Other Plots
Layer 4: Graphics Macros
Rect Grid - Circle - Arc - Frame - Box - Label ... Other Macros
Layer 3: Graphics Primitives
Move - Draw - Ginit - Scale - Pen ... Other Primitives
Layer 2: Graphics Device Drivers
HPGL - IBM PC - TEKTRONIX - DEC VTxx ... Other Devices
Layer 1: Graphics Device Interface
CRT - Disk - Digital Plotter ... Other Devices

Figure 1: *Topdown software design*

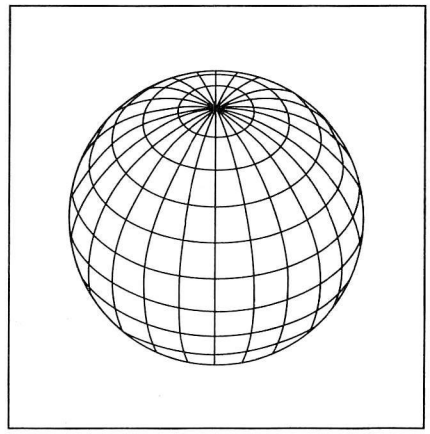


Figure 2: *Sample output from the globe program*

PLICIT NONE eliminates guessing and mistakes automatically when porting code). Also, program loops and structures were not indented (non-indented loops and logic clauses make code more difficult to interpret). Added to this, extended use was made of non-descriptive variable names. (Be sure to spell out variable names when practical to clarify usage and meaning.)

Hard coded logical unit specifics were implemented using WRITE(6,*) . . . which makes pointing to other devices or terminals difficult without modifying all occurrences of output and enter statements. (Global device specifics simplified this process.) Furthermore, the program used improper program context definition. (You should use consistent program and subroutine headers throughout the code so that you can tell at a glance where one module ends and the next one begins.)

Because different machines use different default storage functions, you should pass constants as formal parameters to subroutines or functions. Integer constants passed as formal parameters may default to 16 or 32 bits of storage depending on the compiler or machine.

The minicomputer versions of the program also made excessive use of line numbers. When designing a program that might be ported, you should eliminate line numbers when possible. Not only does the code look less cluttered, but porting to another language is often much easier without line numbers. Finally, DO loops that shared the same CONTINUE termination statement and multiple entry and exit points made the program confusing.

Porting to another language is often much easier without line numbers

In Listing One, most of the problems have been removed while still keeping the code 100 percent compatible with Microsoft Fortran. Note, however, that almost all of the device- and compiler-specific issues are more isolated.

While this program is complete, it does not make allowances for other graphics devices. Therefore, the last changes made before adding the globe program to the Microplots system required expanding the graphics devices supported to include devices in the structure pyramid of Figure 1. To accomplish this goal, the program fragments in Listing Two (page 82) illustrate the further layering of the device initialization process to include other drivers.

The top level of the globe program makes a call to initialize the current graphics device. As in the case of all of the top-level routines, the device specific details are left to lower-level macros, in this case GINIT.

In the fragment of level 3, the subroutine GINIT makes the appropriate call to a device driver pointed to by the symbol DEVICETYPE. The symbol DEVICETYPE is an integer constant set to a number representing one of the many devices defined by the system. A match

The DO's and DON'Ts of Writing Portable Fortran Code

When faced with the task of writing portable software, you may wonder how to assess the quality of your code and a plan of attack. Here are some Do's and Don'ts for system design that may make this task easier. Although these issues are concentrated on Fortran, some of them apply to any programming language or operating system.

The Do's

Define every variable explicitly in each module. Use *Integer*2* or *Real*4* for example. Default word sizes vary from machine to machine.

The ANSI extension of IMPLICIT NONE should be universally adopted. An amazingly large number of debug nightmares and spelling errors can be avoided by this simple statement included at the beginning of each subroutine and function.

Indent all loop bodies and IF clauses.

Use one variable per line and a trailing descriptive clause if possible. The ANSI comment extension allowing the ! trailer is useful for this purpose.

Use the ANSI extended DO-END DO and DO WHILE-END DO to avoid line numbers and continue statements for better code readability.

Use a compiler that allows the MIL-standard ANSI extensions to Fortran 77 if possible.

Use batch or command files for all phases of program compiling and linking. Command-line parameters are usually only remembered by the person writing the module. In time, even the original author may forget which switches to use!

The Don'ts

Do not use computed GOTOs.

Do not use the DIMENSION statement (explicit typing of variables should be used).

Do not use non-integer parameters in *do* loops or shared termination statements.

Do not use alternate RETURN statements.

Do not use non-descriptive variable names such as XXX and YYY.

Do not use long subprograms performing many tasks with multiple exits.

Do not pass constants as formal parameters to subroutines.

Do not embed compiler or machine-dependent directives in source code.

—J.L.B.

(continued from page 30)
is searched for in the body of the IF-THEN structure and the specific lower-level driver is called.

In level 2, the CPU and compiler-specific code for the various devices is selected. If and only if the current device happens to be of the IBM PC variety, the IBMPCINIT driver is called. IBMPCINIT contains specific references to the graphics library routines supplied with Microsoft 5.0 Fortran. If and only if the current device is of the VAXstation variety, the VAXSTAINIT driver is called. This driver contains symbols and library references known only to the VAX compiler and CPU.

As illustrated by this test case, the code least portable is isolated to "well-defined" driver modules at the lowest levels of the graphics library. As a result, the graphics routines can be ported to virtually any CPU with minimal effort. In addition, an unlimited number of additional graphics devices and plotting algorithms can be added to the Microplots system.

This approach is not without some minor compromise in performance and flexibility in a few areas. However, when faced with the options of porting code riddled with CPU and compiler-specific options or changing a few low-level drivers, the compromises can save many man months of coding and debug efforts.

It is interesting to note here that the top-down approach used for the graphics design can also be used in other scientific application areas such as file, signal processing, and I/O libraries.

Future Considerations for PC Fortran 77 Development

Efforts of most Fortran suppliers have made porting large-scale programs to the PC quite practical. However, Fortran's use on PCs for development of new software may continue to be phased out in the future. Until more system services and features are added to the language and supported (will Fortran 8x ever be released?), mixed-language calls to Assembler or C will continue to be a necessary evil. As C becomes more object oriented and Fortran standards continue to stagnate, the notion of using Fortran for reasons other than performing complex math functions may prove impossible to defend in the future.

DDJ

(Listings begin on page 80.)

Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

Persistent Objects in Turbo Pascal

Create a storable object type to share objects between applications

Scott Robert Ladd

There are four fundamental features of any true object-oriented programming language: data abstraction, encapsulation, inheritance, and polymorphism. These are the core concepts necessary for object-oriented programming. However, they aren't the only concepts that are useful to the object-oriented programmer.

With most programming languages, objects are created, manipulated, and destroyed within the context of a single program. Each program is a community of objects that do not communicate with other communities of objects. Put yourself in the place of an object. Think of inter-object communication as a network of roads and each program as a city. You can go anywhere inside your own city, but there are no roads leading to other cities. This is frustrating, because it limits the ability of different programs within a system to share objects.



Supporting Persistence

Turbo Pascal, C++, and Smalltalk do not directly support the saving of objects from one invocation of a program to another. This is like building a car every morning, driving it to work, and then having it disintegrate when you get back home at night. Every time you commute to work, you need to build a new car. Similarly, whenever you create an object in a program, it exists only until the program's work is done. Every time the program starts, it has to begin by building new objects.

Bertrand Meyer incorporated the concept of "persistent objects" into his Eiffel programming language. A persistent object can be stored external to a program, usually in a file. More than one program can share the same persistent object, and a program can save its objects for future executions.

Eiffel implements persistent objects via a special base class called *STORABLE*. Objects of a class derived from *STORABLE* have *store* and *retrieve* methods. When an object is stored, its entire data structure is written to a specified file. Later, that entire structure can be loaded into an object of the same class.

Does having built-in support for persistent objects make Eiffel a superior object-oriented programming language? In one sense, the answer is yes: Anything a programming language can provide for you is a definite advantage over having to develop a facility on your own. However, persistence can be accomplished in other programming languages with a minimal amount of work and the knowledge of object formats.

Persistence with Turbo Pascal

The Turbo Pascal object implementation is relatively simple when compared to Smalltalk and C++. Listing One (page 84) shows an example of a simple Turbo Pascal class hierarchy. *Shape* is an abstract base class, which defines the common characteristics of all *Shape* classes. Derived from *Shape* are two classes, *Box* and *Ring*. *Box* defines a rectangular shape, and *Ring* defines a circle. The *Draw* method is polymorphic, by virtue of the *VIRTUAL* keyword.

Listing Two (page 84) is a short program which uses the polymorphic nature of *Draw* to implement a generic procedure called *DrawShape*. The argument to *DrawShape* is a pointer to the common base class (*Shape*) of *Box* and *Ring*. The call to the *Draw* method in *DrawShape* invokes the implementation of *Draw* for the specific class of the object whose pointer was used as an

Scott is a free-lance writer; he has more than 15 years of experience in a variety of programming languages. Scott can be reached at 705 W. Virginia, Gunnison, CO 81230.

(continued from page 36)
argument. In other words, passing a pointer to a *Box* to *DrawShape* will cause the *Box.Draw* method to be called.

Making Turbo Pascal objects persistent requires that we understand how objects are stored in memory. A Turbo Pascal object is basically the same thing as a record: The instance variables of

Persistence can be accomplished in other programming languages with a minimal amount of work and knowledge of object formats

an object are stored contiguously in memory, aligned on 16-bit word boundaries. A derived object class will contain all of the members defined for its base class, plus all of the members unique to itself. *Shape* defines a single instance variable, *Color*. Because *Box* is derived from *Shape*, it inherits *Color*. You can think of *Box*, for persistence purposes, as the Pascal *RECORD* structure shown in Example 1.

The obvious approach to writing an object to disk is to use the same method used for writing a record. Were *Box* a record as shown in Example 1, a *FILE OF Box* could be declared to hold *Box* objects. Alas, a file component type in Turbo Pascal cannot be an object type; *FILE OF Box* generates an error to this effect.

An untyped file allows us to solve our problem. Listing Three (page 84) employs an untyped file and the *BlockWrite* and *BlockRead* functions to store and retrieve *Ring* and *Box* objects. The *Reset* and *Rewrite* functions declare the file record length to be one, letting us write any number of bytes to the untyped file. When we read and write an object, the *SizeOf* function is used to get the size of the object.

```
BoxRec = RECORD
     BEGIN
          Color : INTEGER;
          UpperLeftX, UpperLeftY : INTEGER;
          LowerRightX, LowerRightY : INTEGER;
     END;
```

Example 1: *Representing the* Box *object as a Turbo Pascal record.*

An object that has virtual methods must be initialized with a *CONSTRUCTOR* call before it can be loaded with an object value from disk. The *CONSTRUCTOR* assigns the address to virtual function pointers for the object. The values of the instance variables can then be updated via a call to *BlockRead*.

Using individual calls to *BlockWrite* and *BlockRead* will store the instance variables of an object in a disk file. This isn't the best method, though. For example, let's say that we wanted to create a function which writes any type of *Shape* to a file. An obvious first attempt at such a function might look like that shown in Example 2.

It looks good, but it won't do what you might expect. A *Shape* is defined with only one integer instance variable, so *SizeOf(Shape)* is only two. A *Box* has 10 bytes of instance variables, and a *Ring* has 8 bytes. So, passing the address of a *Box* (for example) to *WriteShape* will result in having only the first 2 bytes of the *Box* stored. Obviously, this is not going to work. A better system is to use methods to store and retrieve objects in files. By making these methods polymorphic across a class hierarchy, generic functions such as *WriteShape* can be created.

Listing Four (page 84) shows new versions of the *Shape*, *Ring*, and *Box* classes. The *Shape* object type defines a pair of virtual methods called *FWrite* and *FRead*. The *Box* and *Ring* classes define specific versions of these methods, which write and read the corresponding object type into the specified file. Listing Five (page 86) shows how these functions can be used in place of the *BlockRead* and *BlockWrite* calls used in Listing Three. Using these new methods, a generic *WriteShape* procedure would look like that shown in Example 3.

What about building a Turbo Pascal class (object type) that emulates the function of the *STORABLE* class included with Eiffel? Because Turbo Pascal supports only single-inheritance — where a class may have only one base class — *STORABLE* would have to be the

```
PROCEDURE WriteShape(f : FILE; s : ^Shape);
BEGIN
     BlockWrite(f,s,SizeOf(s^));
END;
```

Example 2: *A first attempt at writing a procedure to store a persistent object.*

```
PROCEDURE WriteShape(f : FILE; s : ^Shape);
BEGIN
     s->FWrite(f);
END;
```

Example 3: *A generic* WriteShape *procedure to store persistent objects.*

only base class. Using a *STORABLE* base class with the *Shape* classes would change their definitions only slightly.

Consider the *Storable* object type defined in Example 4. The *Shape* object type and the object types derived from it inherit the basic I/O functions from *Storable*. Every object type for whom *Storable* is an ancestor defines its own implementation of the *FWrite* and *FRead* methods.

Another advantage of read/write methods customized to an object type can be seen with objects that represent dynamic data structures. Let's say that you've created a linked-list object type: Each linked-list object will have a pointer to the first node in the list; the other pointers and the actual data will be stored in dynamic memory. Simply writing a linked-list object directly with a *BlockWrite* statement will store the pointer only to the first element in the list.

```
Storable = OBJECT
     BEGIN
     PROCEDURE FWrite(f : FILE);
     PROCEDURE FRead(f : FILE);
     END;

Shape = OBJECT (Storable)
     BEGIN
     Color : INTEGER;

     PROCEDURE Draw; VIRTUAL;
     END;
```

Example 4: *A storable base class (object type) that emulates the features of the Eiffel* storable *class.*

Obviously, this pointer will be meaningless in another execution of the program (because addresses will change). Even worse, the other nodes in the list will be lost when the program terminates. Class-specific read/write methods can solve this problem by writing the data in the nodes of the linked list to the disk file in the same order as they appear in the list. Then a linked list is read from disk, and the dynamic tree can be recreated from the node data.

Conclusion

Persistent objects are useful in restoring the state of objects within a program from one run to the next. This is particularly important for objects that are created on-the-fly. In addition, persistent objects allow different programs to share common objects.

As object-oriented technology unfolds, persistent objects will play an increasingly important role. The benefits will be seen not only in programming, but in applications such as object-oriented databases and, ultimately, in operating systems.

The greatest benefits to programmers will come when persistent objects can be shared between programs written in different languages. Wouldn't it be nice to grab a Pascal object from C++?

A common specification for the creation and management of persistent objects, such as that being proposed by Microsoft, will be needed. As you have seen in this article, it is relatively easy to implement persistent objects in Turbo Pascal. With slight modifications, the ideas presented here can be applied to both Smalltalk and C++.

Source Code Availability

As a service to our readers, all source code is available on a single disk and on line. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Drive, Redwood City, CA 94063, or call 800-356-2002 (inside Calif.) or 800-533-4372 (outside Calif.). Specify issue number and disk format. Code is also available through M&T's Telepath on-line service (via TYMNET) and through the DDJ Forum on CompuServe (type GO DDJ).

DDJ

(Listings begin on page 84.)

Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

Fast Search

File access in the fast lane

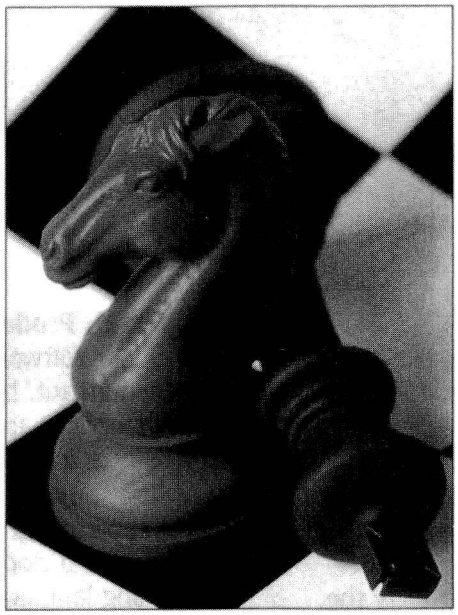
Leon Campise

This article presents FASTSRCH, a Basic function that scans a data file and returns an array containing the record numbers of all records that match a specified key. FASTSRCH applies a couple of basic concepts to attain very fast sequential access to data files: First, it reads and processes many records at a time using Basic's binary file I/O, yielding fewer Disk Read Requests to DOS. Secondly, it uses Basic's *INSTR* function to "instantly" scan for records meeting the desired criteria.

The idea is to scan the data file as fast as the disk drive can access it. This is not attained in the traditional record processing as there is a great deal of overhead that goes along with retrieving each record, comparing the key against a specific value, and iterating a loop counter.

We use FASTSRCH in almost every case where we used to scan files sequentially. The areas where you see the most dramatic gains are with smaller records. However, speed increases with almost any record length. This is especially true when incorporating *InstrASM* in lieu of Basic's *INSTR*. (See the accompanying text box, "Beyond FASTSRCH.")

Leon is the senior managing engineer for a software developing firm specializing in medical practice management systems. Leon can be reached at Dental Plan Inc., 3633 Broadway, Garland, TX 75043.



On transactional files having record lengths less than about 100 bytes, the speed increase is amazing. For example, one application, which finds and displays all transactions for a certain customer, searches a summary transaction file that has a record length of 32 bytes. I benchmarked the old code on a large data set which took almost 1 minute, 20 seconds. Using FASTSRCH, the file was processed in 6 seconds!

Calling FASTSRCH

FASTSRCH is easy to call in a program. First, you dimension an array that will hold the found record numbers, then open the file to be searched, and then execute the call. When FASTSRCH returns control, the array has all matching record numbers loaded and is ready to go!

Figure 1 could be from a program that scans a transaction file and displays all records that belong to a particular customer. This example shows how it might be done in a traditional manner. In Figure 2, FASTSRCH is used for the scan. The first difference when using FASTSRCH is that we must set the maximum number of records we can retrieve in order to dimension the array that will receive the selected records' record numbers. The next difference is we open the file in binary access mode instead of random access mode. This allows FASTSRCH to process many records at once. But the greatest difference in the calling program's logic comes after the call to FASTSRCH. Here, we close and reopen the file in random access mode. Then, instead of looping through all records in the file, searching for matching ones, FASTSRCH has already identified the records we need, and hence, we can access the data directly. The seven parameters to pass to FASTSRCH are listed in Table 1; Table 2 lists the returned values.

Examining FASTSRCH

FASTSRCH (Listing One, page 90), which was compiled with Microsoft Basic 7.1, calculates and loads as many records as it can into a large file buffer.

```
OPEN "TRANSACT.DAT" FOR RANDOM AS #1 LEN = LEN(T)
FOR I% = 1 TO NumRecs%
        GET #1, I%, T
        IF KeyVal$ = T.CustNum THEN CALL Print1Rec(T)
NEXT
```

Figure 1: *Scanning a file without FASTSRCH*

```
DIM FoundRecs%(MaxToFind%)
OPEN "TRANSACT.DAT" FOR BINARY AS #1
NumFound% = FASTSRCH(1, LEN(T), 7,' KeyVal$, 1, LastRec%, FoundRecs%())
CLOSE #1
OPEN "TRANSACT.DAT" FOR RANDOM AS #1 LEN = LEN(T)
FOR I% = 1 TO NumFound%
        GET #1, FoundRecs%(I%), T
        CALL Print1Rec(T)
NEXT
```

Figure 2: *Scanning a file using FASTSRCH*

(continued from page 42)
Then it searches each byte in the buffer for a match on the key. When it finds a match, it uses modular arithmetic (remainder after division) to find out if the matched data is in the key's location.

Parameter	Function
BuffFile%	The file handle used in the OPEN statement.
RecLen%	The fixed length of the data record.
KeyLoc%	The starting byte of the field being matched on.
KeyVal$	The value that the key needs to have to be a "match".
StartRec%	The first record to begin searching on.
LastRec%	The record number of the last number to search.
FoundRecs%()	The array that will hold the record numbers of matching records.

Table 1: *Parameters passed to FASTSRCH*

Parameter	Function
FoundRecs%( )	When this array is passed back, it will contain the matching record numbers.
FASTSRCH	Returns the number of matches found.

Table 2: *Values returned by FASTSRCH*

If it is, it calculates the physical record number of the matched key's record and adds it to the array of matching records; otherwise, it continues to search the rest of the buffer. When the entire buffer is searched, it loads the next group of records into the buffer and starts over.

In detail, FASTSRCH starts out by creating a 10,000-byte file buffer in the far heap. My consultant, Bill McGill and I have found that a buffer larger than this doesn't seem to speed things up much. Next we perform some initialization.

Note that we check to see if the key that we will be matching on starts on the last byte of the logical record length (even though this would rarely be the case). If it does, we set the key location compare variable to zero instead of the key location. (Because we will later be using modulo arithmetic to verify that a match is at the key location.) If the key location is the last byte of the record, the remainder will be zero instead of the record length. We use integer division to calculate the number of whole records that can fit into our file buffer: This is the maximum amount of buffer that we will use.

The outer loop executes once for each buffer full of records we search. The first thing we do is calculate the byte in the file that is the first byte we will load into the buffer. Then, starting at that byte, we retrieve the data into the buffer. Next, if this is the last buffer and it is not full, we restrict the number of bytes to search.

The middle loop executes each time we find a matching record in this buffer. The innermost loop exists because Basic's *INSTR* will return a "false match" if there is a match not in the key's location. This loop, therefore, is needed

(continued from page 44)
to weed out the false matches. When *INSTR* finds a match, we check to make sure it isn't past our logical buffer length. If it is, or there just weren't any matches at all, we exit the loop. If there was a match then, the *LOOP UNTIL* makes sure that we are on the key position before exiting the loop and adding this record to our other found records. Otherwise, we loop again, continuing from the byte where we left off.

When we fall out of this inner loop, we check to see if there was a match, and if there is room for it in our array. If so, we calculate the physical record number and add it to the array.

This middle loop will continue until there are no more matches in this buffer. The next logical record number is calculated and we loop again if we are not past the end of the file. Finally, we set the number of records found to FASTSRCH%.

Modifying FASTSRCH

The FASTSRCH function listed with this article is a bare-bone version to use for discussion. Once you understand

(continued on page 49)

Beyond FASTSRCH

FASTSRCH has a few limitations that can be overcome by using customized, low-level searching routines in place of *INSTR*. Performance gain can be lessened when searching long records. Because FASTSRCH uses Basic's *INSTR* to find a match, every byte in the buffer must be searched sequentially. When the records are short, there are not many extraneous comparisons. But when records are long, say 256 bytes, up to 255 extraneous compares are being made for each record. This problem becomes compounded when the value of the key being compared matches in other places in the records. These "false" matches can further degrade performance.

A solution to the problem is to create a smarter *INSTR* function that knows at which position in the records the key is located, and checks only this key data. For this to execute as quickly as possible, the core of the search must be written in assembler. We have written two routines: *InstrR* (see Listing Two, page 90) written in Basic and compiled with Microsoft Basic 7.1, is called by FASTSRCH, and *InstrASM* (Listing Three, page 90), written in assembler and assembled with MASM 5.1, is called by *InstrR*. *InstrR* basically does the setup for *InstrASM*, which actually performs the search.

In order to implement *InstrR*, FASTSRCH needs to be simplified slightly. Because *InstrR* only returns matches on the key (no false matches) then the entire inner loop of FASTSRCH can be replaced with the lines in Figure 3(a). Also, the formula for calculating the matching record's actual record number must be simplified as shown in Figure 3(b).

Using Conditionals Other than "Equal to"

Another limitation of the FASTSRCH when using *INSTR* is its ability to select records based only on matching a key value exactly. However, when using *InstrASM*, we have control over which conditional we use to make selections. Instead of using JE (jump if equal), we can use JA (jump if above), or JB (jump if below) any other conditional. Changing the one jump statement is all it takes to make this modification, which adds a tremendous amount of flexibility to our searching capabilities.

— L.C.

(a)

```
Match% = InstrR(Match% + 1, FarBuff(), KeyVal$, RecLen%, KeyPos%)
IF Match% > BuffLen% then Match% = 0
```

(b)

```
FoundRecs%(NumMatches%) = Match% + CurRec% - 1
```

Figure 3: *Implementing* InstrR

(continued from page 46)
the concept behind it, you can customize and enhance FASTSRCH to improve performance, flexibility, and presentation. Here are a few considerations:

- If the file being searched can grow to have more than about 32,000 records, then a *long* integer array should be used, and FASTSRCH itself may need to return a *long* integer.
- If records longer than about 100 bytes are to be searched, or if there will be many false matches (matches not in the key's location) then a smarter, more sophisticated version of *INSTR* can be used to enhance performance. We have written such a routine in assembler called *InstrASM*.
- If you need more flexible searching capabilities, such as greater than or less than a specified value, you can use a slightly modified *InstrASM*.
- FASTSRCH is very conducive to having a "percent finished" indicator displayed on the screen without slowing things down. Simply set up the window before calling FASTSRCH and add the following line as the first statement inside the outer *DO* loop: *LOCATE Row%, Col%, 0: PRINT USING "###";(CurRec% / LastRec%) * 100;*
- If it is likely that the number of matches is not predictable then you can have FASTSRCH return "-1" to indicate the capacity of the dimensioned array was exceeded. Then the data can be processed and control could be passed back to FASTSRCH to process the rest of the file.

When it comes to accessing data files quickly without the hassle of maintaining sophisticated linked lists, or using database engines, FASTSRCH can be the answer to the "Searching, Please wait . . . " blues.

Source Code Availability

As a service to our readers, all source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Drive, Redwood City, CA 94063, or call 800-356-2002 (inside Calif.) or 800-533-4372 (outside Calif.). Specify issue number and disk format. Code is also available through M&T's Telepath on-line service (via TYMNET) and through the DDJ Forum on CompuServe (type GO DDJ).

DDJ

(Listings begin on page 90.)

Vote for your favorite feature/article.
Circle Reader Service **No. 4.**

A Generic One-Pass Assembler

Symbol management is the key to one-pass assembly

William E. Ives

The one thing most programmers are interested in when developing code in assembly language is fast assembly turnaround time. A one-pass assembler facilitates this by avoiding the delays associated with multipass assemblers, and thus is able to out-perform them. This performance increase has some cost, the foremost being optimization. But even this can be minimized by proper symbol management.

This article describes one approach to a generic one-pass assembly. The ideas presented can be applied to most of the current assembly languages.

Symbols

A symbol is an alphanumeric shorthand used by programmers to reference a value that the symbol is tied to at some time during assembly. As an example, consider the symbol *THERE* in the following code fragment:

```
        JMP THERE
        . . .
THERE ADD #1,D0
```

The assembler must interpret the value of *THERE* as being the address of the *ADD* instruction. It must then generate the machine code for the *JMP* instruction with the resolved address. Notice that *#1* and *D0* are not considered symbols because their values or meanings are implied within the assembly language. In this case *#1* means the immediate value of one, and *D0* means data register zero.

As shown above, a symbol can be tied to an address by a line label. Most assemblers also provide a method of defining symbols through an assembler *EQUATE* pseudocommand.

```
ONE EQUATE 1
```

Here, the *EQUATE* does not assemble to any machine code, but rather instructs the assembler to tie the value *1* to the symbol *ONE*.

Once the symbols are either tied to values or resolved, the job of the assembler is to use these values to generate machine code for those instructions whose operands reference symbols. For example, the *JMP* instruction shown earlier would normally have a base opcode followed by a target address within its machine code. The assembler fills in the target address portion of the machine code once the symbol *THERE* is resolved.

Forward References

Assemblers traditionally pass through code using a location counter to keep

```
0000   A6 8D 0003          LDA FORWARD,PCR  ; 16-bit offset
                           SHORT
0004   A6 8C 00            LDA FORWARD,PCR  ; 8-bit offset
0007   01       FORWARD    FCB 1    ; Force constant byte
```

Figure 1: *Example of the* SHORT *directive to generate an 8-bit offset*

```
                           ABS_SHORT              ; set default to absolute word
0000   2038 000A           MOVE.B D0,LABEL
                           ABS_LONG               ; set default to absolute long
0004   2039 0000           MOVE.B D0,LABEL
       000A
000A   01        LABEL DC.B 1  ; define constant byte
```

Figure 2: *Changing the default addressing mode on the 68000*

```
0000   6004                BRA.B    TARGET  ; 8 bit offset of 4
0002   6000 0002           BRA      TARGET  ; 16 bit offset of 2
0006   6000 FFFE  TARGET   BRA      TARGET  ; 16 bit offset of -2
```

Figure 3: *Making a branch command deterministic using the* .B *size specifier*

William is a software design engineer in the Colorado Telecommunications division of Hewlett Packard, 5070 Centennial Blvd., Colorado Springs, CO 80919. William's Internet address is wei@hpctdlb.hp.com.

(continued from page 50)
track of the address of each instruction. As each instruction is processed, the location counter is incremented based upon the number of bytes required to hold the machine code of the instruction. This forces the sequential location of machine code for each instruction with no "undefined" bytes between instructions. In addition, it leaves space in the final machine code to go back and fill in resolved symbols such as *THERE*.

This is a problem, however, when the amount of space to be reserved for forward referenced symbols is unknown. This is especially apparent on processors that offer multiple forms of the same instruction. For example, the *JMP* instruction might come in either form: 16-bit or 32-bit target address. The question is how much space is left in the machine code if the assembler does not yet know the value of the target address (*THERE*).

Multipass assemblers address this dilemma by resolving symbols on the first pass through, adjusting instruction addresses appropriately, then generating machine code on additional passes. With this approach, most multipass assemblers handle both the forward references and machine code optimization. Although this method generates optimal machine code, it is often time-consuming and thus tedious for the programmer who is looking for a fast turnaround time.

A one-pass assembler addresses this same problem by passing through the code only once, leaving space for the forward referenced symbols, then filling in these spaces as the symbols are resolved. This process is called "back-patching." By making the assembly language deterministic with assembler defaults or directives, the one-pass assembler knows how much space to leave for forward referenced symbols. This may not result in optimal code, but does result in improved assembly time.

As with any multiple module assembly, a linker must be used to create the final assembly program

The directives or defaults needed to make this possible depend heavily on the actual processor and its addressing modes. For example, the 6809 constant offset program counter for the relative mode has an 8-bit and a 16-bit form. This can be made deterministic by defaulting to the larger size (16-bit) and allowing the user to force the 8-bit offset by using an assembler directive (*SHORT*). The code in Figure 1 demonstrates this.

An example for the 68000 involves the absolute addressing mode, which can be either a 16-bit or 32-bit address. Whenever a forward referenced symbol is used for these modes, the current assembler default is used. This default size can be changed from *word* to *long* by the directives shown in Figure 2.

The 68000 branch commands must also be deterministic because they come in byte and word forms. This is done by defaulting to the word offset unless the instruction is qualified by a size specifier of *.B*, as shown in Figure 3.

These limitations might seem unreasonable at first, but the default assembler settings can be used so that minimal effort is required on the part of the programmer. The programmer can use the directives to override the defaults for optimization; this need be done only for forward references because backward references can automatically be optimized. This approach is not at

(continued from page 52)
all uncommon in many of today's commercial assemblers. For example, those familiar with Microsoft's Macro Assembler will recognize the directive, *jmp short there*, which tells the assembler that the *jump* reference is within the current code segment.

Symbol Management

Once an assembly language is made deterministic, symbol management becomes the key to the assembly process. First, each line is parsed, and any symbols are extracted from it. Then a machine code size determination is made based on assembler defaults and directives. If all of an instruction's operand symbols are resolved, the machine code can be immediately generated. Otherwise, all information associated with the instruction and its unresolved symbols is saved (in an operand reference list) for later assembly through backpatching. In either case, the location counter is incremented by the size of the machine code, and the next source line is similarly processed.

The symbols for each line are processed immediately after they are parsed from the line. The two types of symbols are the line label and any operand symbols. Each type of symbol is associated with a list: A label with the resolved value list and an operand symbol with the reference list. The line label is resolved to the current value of the location counter. The label and its value are added to the label list so that future references to the label can be resolved. Then, any previous references to the line label are taken from the operand reference list. These are resolved, and machine code generation is done for each of their instructions through backpatching. A flowchart of this is shown in Figure 4.

The operand symbol management

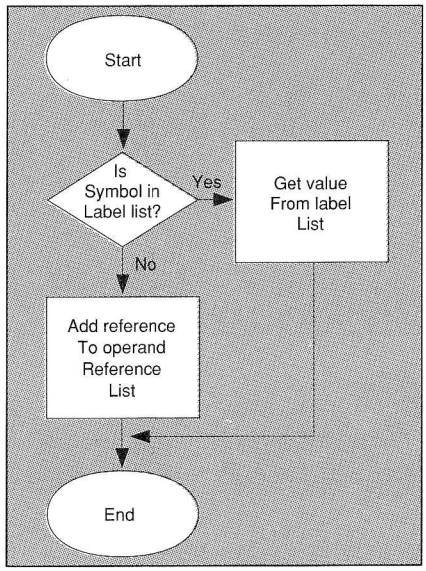


Figure 4: *Operand symbol management*

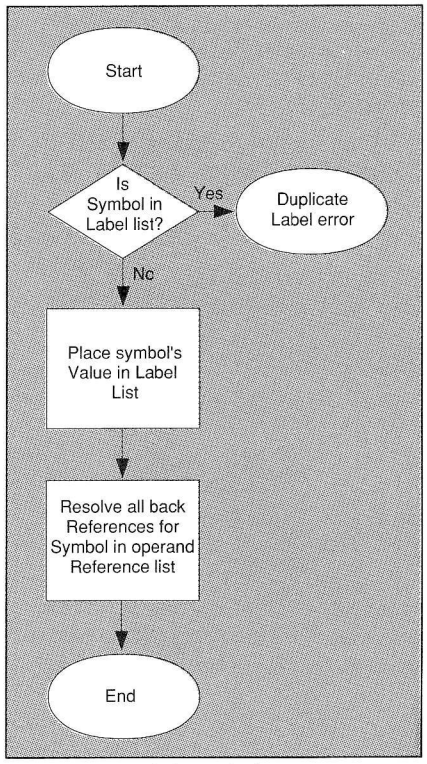


Figure 5: *Label symbol management*

(continued from page 54)
was described earlier. A flowchart for this is shown in Figure 5. An example of assembling a generic piece of assembly code using these two flowcharts is shown in Figure 6.

By making the assembly language deterministic with assembler defaults or directives, the one-pass assembler knows how much space to leave for forward-referenced symbols

Note that at any given time the operand reference lists contain only those forward references not yet resolved. In this respect, it does rely on memory being available for expansion, but the list size is always at a minimum for the code being assembled. Again, this is the trade-off of space versus time.

Multiple Modules

The above management model can easily be expanded to support multiple module assembly. This is done by introducing a global directive to export symbols from a module and an external directive to import symbols. In addition, two new lists are added: A global label list and an external reference list. These are similar to the label list and operand reference list, respectively.

The new flowcharts are shown in Figures 7 and 8. The major difference to note in the label symbol management flowchart (Figure 8) is the order in which the various label lists are searched. By searching the external and global lists first, this flowchart precludes the possibility of mistakenly placing a global label in the local label list. This order of searching also provides the scoping rules between local and global symbols.

The operand symbol management flowchart (Figure 7) shows another point of interest. If a global label has not been resolved before it is referenced as an operand symbol, the reference is added to the local operand reference list. This is done because a global symbol must be resolved before the end of a module. Thus, any references to it can be treated just like local references.

Linking

As with any multiple module assembly, a linker must be used to create the final

Address	Label	Command	Operand
L1	Top	Jmp	Forward
L2		Move	D0. Data
L3		Lea	A0. Top
L4	Forward	Clr	Data
L5	Data	Dc.b	1

Line processed	Label list	Operand reference list
L1	Symbol / Value: Top / L1	Symbol / Ref.: Forward / L1
L2	No change	Forward / L1; Data / L2
L3	No change	No change
L4	Top / L1; Forward / L4	Backpatched L1 Forward resolved; Data / L2 / L4
	Top / L1; Forward / L4; Data / L5	Backpatched L2 & L4 Data resolved

Figure 6: *Assembly example using flowcharts from Figures 4 and 5*

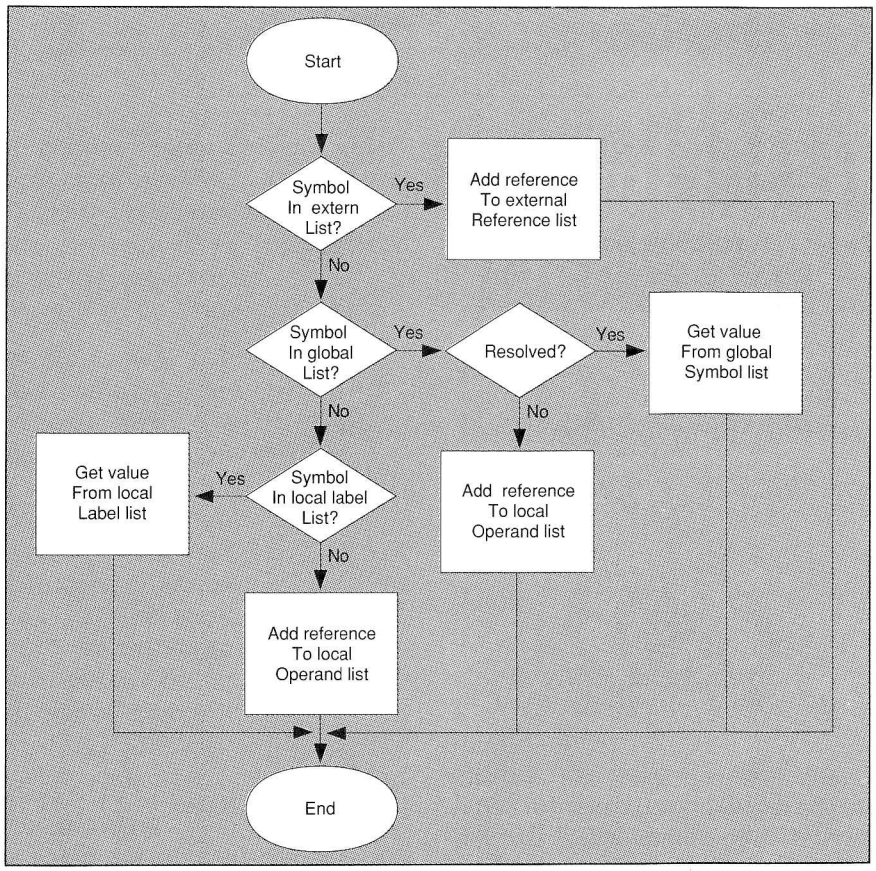


Figure 7: *Operand symbol management flowchart*

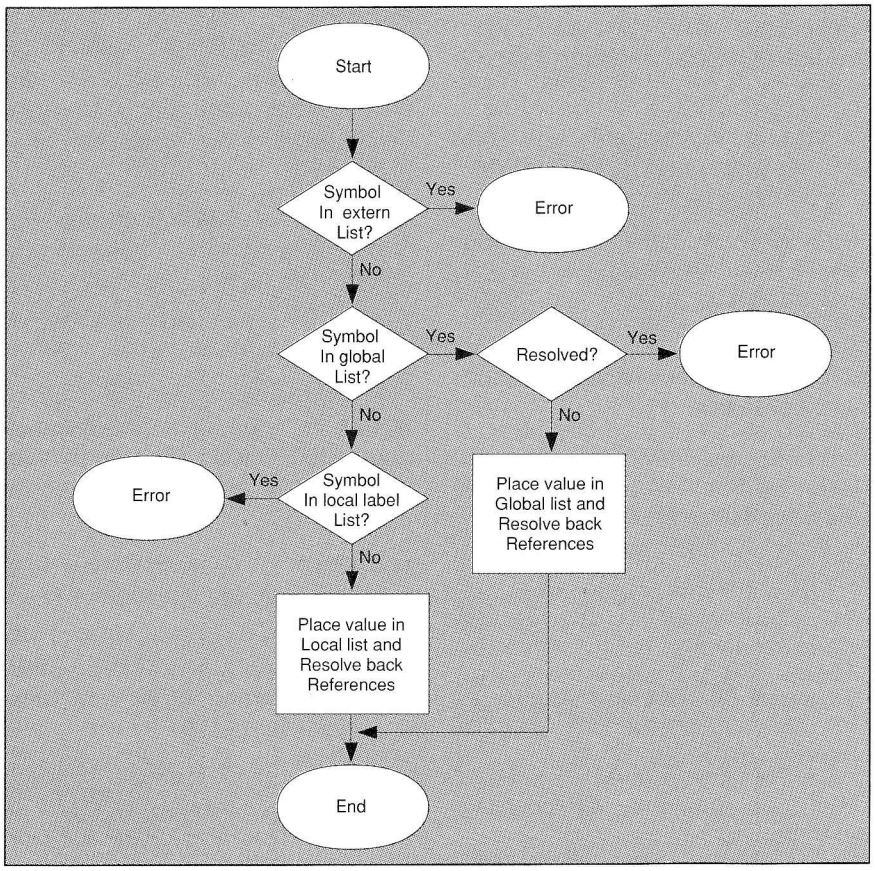


Figure 8: *Label symbol management flowchart*

(continued from page 56)

assembly program. Here, again, the one-pass symbol management approach is used because linking is just a matter of resolving external references across modules and generating machine code through backpatching. Of course, linking involves much more, but a detailed discussion is beyond the scope of this article.

One feature of linking is relocation, which is not too difficult to add to the current model because it merely involves treating all symbols as relative to the base of the module. Once the module base address is known, all symbols for the module are resolved. Another advanced feature is arithmetic expression support. This can be added by using expression trees in place of the symbols. Resolution then becomes the resolution of the entire tree for each instruction's operand. Unfortunately, each of these adds overhead and slows down the assembler. If these advanced features are used extensively, the trade-off may be worthwhile.

A One-Pass Assembler

Listings One through Nine provide a simple one-pass assembler for a generic assembly language. Listing One (page 92) provides the main driver; Listing Two (page 92) contains the procedures to handle assembly; Listing Three (page 96) is the error handler module; Listing Four (page 97) contains the procedures to handle the instruction set; Listing Five (page 98) is the listing module; Listing Six (page 100) is the math module; Listing Seven (page 100) is the parser module; Listing Eight (page 102) provides the procedures for pseudocommand assembly; and Listing Nine (page 103) is the symbol table module.

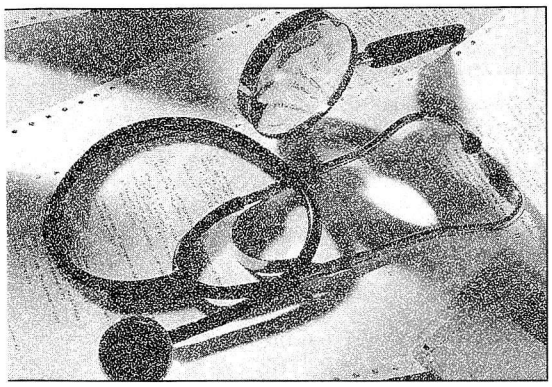
Because of space considerations, all comments have been removed from the code. Fully commented code, further test cases, and an executable version of the assembler are available from *DDJ*. (For more information see page 3.)

This code can easily be modified to build a full-featured assembler by adding an advanced parser, instruction opcode lookup facility, and error trapping. In all, symbol management is the key to one-pass assembly. And, although such an assembler may not create optimal code, it does have the fastest development turnaround time.

DDJ

(Listings begin on page 92.)

Vote for your favorite feature/article.
Circle Reader Service **No. 5.**



Inside Object Professional

A rich set of objects is the key to object-oriented development in Turbo Pascal

Gary Entsminger

Object Professional, from Turbo Power Software, is a splendid library of object-oriented classes for Turbo Pascal 5.5 programmers. It's extensive — consisting of over 50 units, 130 object types, 2100 documented procedures, functions, and methods, 1200 additional internal routines, and 1600 pages of text and examples neatly divided into 3 manuals.

Object Professional includes complete Turbo Pascal and assembly language source code so you can modify, append, and derive to your heart's delight. If you're a Turbo Pascal 5.5 programmer developing applications that need sophisticated screen management, data input/output, or TSR routines, check out this package. It could save you months of development time.

Pick an Object

Object Professional includes objects for screen, keyboard, and mouse handling; windows, menus, pick lists, and directories; text editing and viewing; data entry and code generation for data entry screens; printing typefaces, control sequences, and forms; data objects ("container classes"), lists, sets, and so on; handling longer than 255 character strings; interfacing to DOS; creating TSRs that automatically swap to EMS or disk; memory allocation and deallocation; keyboard macros; BCD arithmetic; and more. The list of objects really goes on and on.

Much of Object Professional is a translation of Turbo Professional, Turbo Power's state-of-the-art library of non-object-oriented subroutines for Turbo Pascal 4.0 and 5.0, and much of Object Professional is new. Container classes, a swapping EXEC function, improved windows and mouse support, large arrays that can use RAM, disk, or EMS, text and binary file viewing, complete window-oriented text editing, and several utilities for making data entry screens and menus have been added. It would (literally) take a dozen pages just to list the new procedures and functions. Instead of doing that, let's take another approach toward examining Object Professional. One that focuses on the object-oriented structure of the library.

Figure 1 shows a portion of Object Professional's object hierarchy. The hierarchy is well-thought out and is thorough. Abstract objects (such as *AbstractField*, *AbstractFrame*, *AbstractWindow*, *AbstractSelector*, *AbstractHelpWindow*, and *AbstractArray*) are built into the library making it easy to derive new objects from Toolbox objects and to implement polymorphism.

Root is the ancestor of all the objects in the library except *ColorSet*. The highest level objects (and the most complex) are the most indented in the hierarchy. Note well that Object Professional is primarily built up from data structures. Various lists (single, double, and so on), arrays, and windows, for example, are ancestors for many of the objects in the toolbox (see Figure 2). A *TextEditor*, for example, is a complex object that rivals the Turbo Pascal editor in power and flexibility. It's very complete — with word wrapping, cursor positioning, extensive file I/O, block reads, writes, copies, moves, and its own editing and command windows. A text editor can be incorporated into an application very easily. Listing One (page 108) shows how little code is needed to create an editor window, read a file, edit the file, and save it.

In addition, the *TextEditor* offers two sets of programming hooks that allow you to completely customizes *TextEditor* objects for specific applications. One set takes procedure pointers as parameters, thus allowing you to specify which procedures will be called when a string, file name, or yes/no answer needs to be entered by a user.

Another set lets you implement your own commands — to invoke an editor/window and move the cursor to a specific location in a text buffer, for example. You could first run a program (such as a compiler) and then use the results from that program (and this set of commands) to go to the position in the file that contains an error, to automatically search for and replace text, and so on.

Gary is a writer, programmer, and a consultant. He is the co-author (with Bruce Eckel) of Tao of Objects *to be published later this year. Gary can be reached at Rocky Mountain Biological Lab., Crested Butte, CO 81224, or through CompuServe: 71141,3006.*

```
ColorSet
Root
        PointerStack
        WindowStack
    ByteStack
    StaticQueue
    SingleListNode
        TextField
        PrintMode
        ...
    DoubleListNode
        HeaderNode
        ShadowNode
        HotNode
        MenuItemNode
        AbstractField
            SelectField
                EntryField
                    StringField
                    ArrayField
                    PrintField
                        StringPrintField
                        CharPrintField
                        ...
                        LineField
                            ShadedField
    SingleList
    DoubleList
        CircularList
            MenuItemList
    BitSet
        LargeBitSet
    StringDict
        StringSet
    StringArray
    AbstractArray
        RAMArray
        EMSArray
        VirtualArray
        ...
        OPArray
    DirEntry
    IDStream
        DOSIDStream
            BufIDStream
                    Library
        MemIDStream
                MemLibrary
    CommandProcessor
    CommandPacker
    Cloner
    ScrollBar
    ScreenRect
    LoadableColorSet
    PackedWindow
    VitScreen
    LineEditor
    SimpleLineEditor
    AbstractFrame
        Frame
    AbstractWindow
        RawWindow
        SubMenu
        StackWindow
            CommandWindow
                Browser
                Memo
                    MemoFile
                        TextEditor
                Menu
                PickList
                    DirList
                    PathList
                    AbstractHelpWindow
                            PagedHelpWindow
                            ScrollingHelpWindow
                AbstractSelector
                    Selector
                        ScrollingSelector
                    EntryScreen
                        ScrollingEntryScreen
    BasePrinter
        DOSPrinter
        FlexiblePrinter
            BiosPrinter
    Printer
    Report
```

Figure 1: *Subset of the Object Professional object hierarchy.*

(continued from page 62)

In addition, the text editor object, like many of the objects in Object Professional can be made to automatically detect whether a mouse driver is currently loaded, and if so, use a complete set of mouse commands for moving, scrolling, controlling events, and so on. You can also create mouse command windows (more on command windows later) and incorporate them in TSRs. This can be accomplished entirely by using objects contained in the toolbox.

Data Entry

Object Professional's data entry objects are extensive, making it easy to design systems for editing single variables (such as lines or fields) and groups of variables (such as screens). Object Professional's line editor can be used for prompting users for file names, strings, and numbers. You can build entry screens for editing database records, spreadsheets, configuration files, command codes, and so on.

The fundamental structure of Object Professional's data entry system is the "abstract field," which is derived from a *DoubleListNode*. Any field derived from an abstract field possesses the facilities for belonging to and manipulating a list. The field hierarchy derived from first the *DoubleListNode* and then the abstract field looks like that shown in Figure 3.

An abstract field isn't associated with any data. Instead it contains the methods for formatting the variables which

```
Root
  AbstractWindow
    StackWindow
      CommandWindow
        Memo
          MemoFile
            TextEditor
```

Figure 2: *Selected children of the* Root *object type*

```
DoubleListNode
  AbstractField
    SelectField
      EntryField
        StringField
        IntField
        . . .
        MultiLineField
        PickField
      PrintField
        StringPrintField
        . . .
        LineField
          ShadedField
```

Figure 3: *The field hierarchy derived from* DoubleListNode *and the abstract field.*

(continued from page 64)
will be displayed on the screen or will be printed. An abstract field is the link between the data entry and printing facilities in the toolbox.

Using Object Professional's data entry objects, a programmer can derive a high-level object, such as an *EntryScreen*, which treats all fields alike, without knowing exactly which type a particular field contains. An *EntryScreen* could then display the contents of a field, for example, by calling the particular field's *draw* method, which decides (or knows) how to display the field. It can display the field itself, call a driver to display it, or whatever. This is a neat implementation of polymorphism at a high and practical level.

Object Professional is primarily built up from data structures

The data entry objects and units are probably the most complex and comprehensive in the toolbox. The manual devotes 250 pages to these units and the source code disks include several useful demos and examples which you'll want to use in order to better understand the Object Professional approach to object-oriented programming.

Figure 4 shows a screen captured from one of the data entry demos, *MakeMenu*. You can use *MakeMenu* to design a menu system interactively. When you're satisfied with the menu system you've designed, *MakeMenu* generates the source code for the menu system for you. *MakeScreen*, another demo included with Object Professional, lets you design data entry screens interactively, and then generates that source code for you. Nifty.

TSRs

One of the most sophisticated units in Object Professional, the OPINT unit, isn't really object oriented — it manages Interrupt Service Routines (ISRs). You can use this unit to create TSR (Terminate and Stay Resident) applications, which can automatically determine the presence of EMS (expanded memory) and/or a RAM disk or hard drive, and swap themselves out of executable memory (the first 640K) accordingly.

OPINT is basically an ISR manager. In short, it keeps track of interrupt handlers via interrupt handles. The *InitVector* function lets you specify a handle

for each TSR (you can have up to 20), and then uses that handle to keep track of things. If you need to restore an interrupt vector, you can call *RestoreVector* with the handle, and OPINT will furnish DOS with the correct interrupt number and the address of the old service routine. OPINT lets you build nearly bulletproof TSRs while programming at a very high level.

The ISR unit also contains routines that help you write large, complex TSRs. These routines allocate and deallocate alternate stacks, allow you to switch to alternate stacks, and call another procedure from inside an ISR, and let you chain to previous interrupt handlers. Best yet, Object Professional includes several demos (with source code) that are not only useful, but can be used to learn the nitty-gritty details of building complex TSRs with multiple, configurable hot keys.

Listing Two (page 108) shows enough code to create a TSR, which automatically swaps itself to EMS or to disk. The kernel that remains in executable memory is only 6K and will seldom need to be more. Because Object Professional's ISR routines swap all but the management kernel out of executable memory, larger programs don't need to use any more executable memory than smaller ones after a swap. The key to keeping the resident part (the kernel) of a swapping TSR small is to keep the main program small because the main program segment's code is always kept in memory.

Command Processing

One of the most useful units in Object Professional is OPCMD, which includes an object called a *CommandProcessor*. This object translates key sequences into command codes — numbers that can represent all the commands in a program. You can then group these command codes into a set of tables that show the relationships between key sequences and commands (or actions). Any object (method, function, or procedure) can then call the *CommandProcessor* which gets a keystroke, scans the appropriate table, and returns the correct command (or action). An object using the *CommandProcessor* doesn't have to know anything about the actual key pressed. It simply acts on the command that's linked to the keystroke.

To get you started, the OPCMD unit defines a set of standard command codes as constants. For example, consider the command codes shown in Figure 5. To put you in control, OPCMD sets aside 55 "user exit commands," *ccUser0* through *ccUser54*. You can assign these commands to a keystroke or a set of keystrokes. This makes it easy to add hotkey links to commands in many of the Object Professional units. For example, suppose you wanted to link the hotkey Ctrl F1 to a hypertext help system. You'd assign it a user exit command, say, *ccUser1*. Then any object using the command processor would automatically recognize the hotkey and return control of the system to the command assigned to the key. By using more than one table, a key can be linked to many different commands, depending on the situation.

Command processing of this type "protects" you from the low-level details of dealing directly with the key-

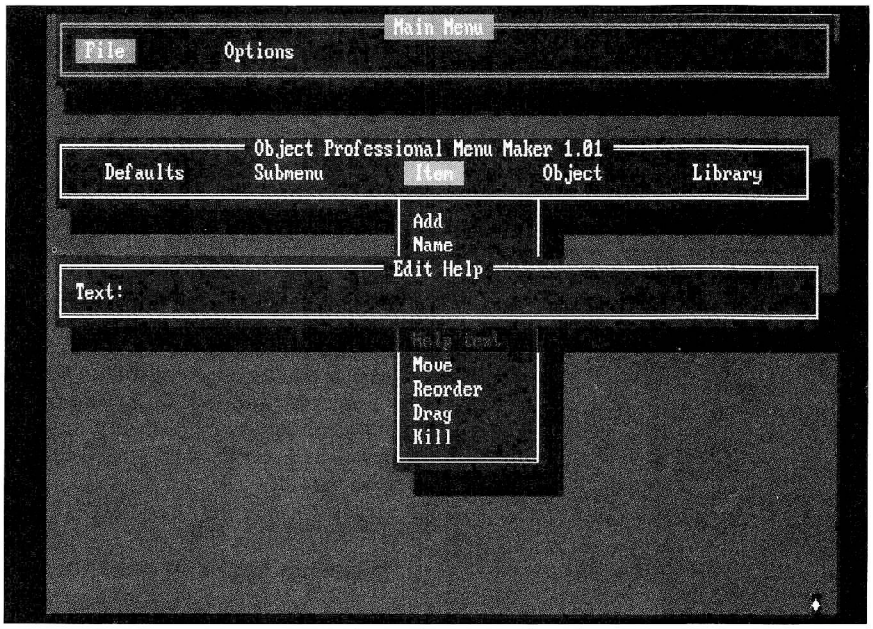


Figure 4: *Screen capture from the* MakeMenu *demo*

board. This saves a little programming time, but more important: Lets you quickly reassign keystrokes to commands, and makes it easy to reconfigure key assignments.

Listing Three (page 108) shows a short program that creates and processes a command window by associating keystrokes with the command codes defined in *KeySet*. Note that a *KeySet* is an array that can hold a virtually unlimited number of key definitions. The array size is determined by the size of the constant *KeyMax*.

Command Windows

Object Professional implements a complete set of windowing objects, which can be used to derive complex and colorful window objects. These include *LoadableColorSet*, *PackedWindow*, *WindowStack*, *AbstractWindow*, *RawWindow*, *StackWindow*, and *CommandWindow*.

The *CommandWindow* object links keyboard and mouse input to the various ways of displaying, viewing, and editing text. A *CommandWindow* essentially takes control of the keyboard (via the *CommandProcessor*), and makes it easy for each window to have its own commands and actions. Text editors, browsers, pick lists, directories, status lines, and so on can be processed within *CommandWindows*. When a *CommandWindow* takes control (via a *CommandProcess*) it doesn't need to know anything about the rest of the program. It's an object, with its own shape, state, and set of commands. Using *CommandWindows* (and a few other Object Professional objects and your imagination) you can create a desktop environment that's comparable to professional environments such as Turbo C++.

```
ccQuit      = 004; { quit processing }
ccMouseSel  = 006; { mouse action }
ccUp        = 012; { cursor up }
ccNewFile   = 112; { read a file }
ccSaveFile  = 113; { save a file }
ccSubNext   = 121; { go to next submenu}
ccSubQuit   = 122; { quit current menu }
```

Figure 5: *Example command codes from the OPCMD unit*

Products Mentioned

Object Professional
Turbo Power Software
P.O. Box 66747
Scotts Valley, CA 95066
408-438-8608
Price: $150
Requirements: IBM PC/compatible
640K RAM
Hard disk, Turbo Pascal 5.5 or later

CommandWindow also includes hooks, which makes it easy to link an error handler and a help system to a *CommandWindow*. The error handling hook is a pointer to a procedure that's automatically called whenever there's an error. The help hook is a procedure for accessing topic-indexed (for example, hyper) help.

Object Professional *CommandWindows* are immediately useful, instructive, and trend setting. Useful because they're powerful and require a minimum of effort to get them up and running. Instructive and trend setting because they're a terrific lesson in thinking about programming OBJECTively and using windows as objects.

The Object Professional approach to command windows is in some ways similar to the approach one must take in order to program in graphic environments such as Microsoft Windows. Command (or message) processing is the name of the game in sophisticated graphic environments. Although the command windows you create with Object Professional can't be readily used in graphic environments, their conceptualization can be.

The Downside?

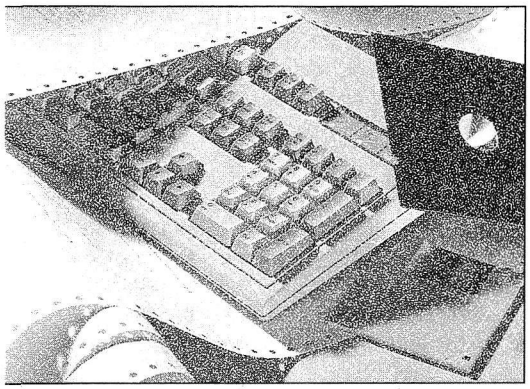
Really, for text processing, Object Professional doesn't have a "true" downside. The procedures, functions, and methods included in this toolbox are tight and fast; many are optimized in assembly language. The source code is complete, and well documented in the listings and in the manuals. The manuals (all 1600 pages of them) are top-notch. The organization, with chapters corresponding to units, is excellent. Each method includes sections on the declaration, purpose, description, example, and "see also." There is usually one, or at most, two methods to a page. The index could (of course) be a lot better! *Nobody* (including the big guys!) spends enough time on indices.

Because there isn't much I don't like about Object Professional, I'll conclude with a couple of (simple!) additional objects I'd like to see. In short, *two more Object Professionals* — one for Turbo C++ and one for Microsoft Windows. A toolkit as powerful as Object Professional applied to an important graphic environment (such as Windows) would be a coup. Are you listening Turbo Power?

DDJ

(Listings begin on page 108.)

Vote for your favorite feature/article.
Circle Reader Service **No. 6.**



Kermit for OS/2 Part I

The learning curve for OS/2 may be steep, but it's worth the price

Brian R. Anderson

I am one of the many DOS curmudgeons who eschewed OS/2 since its introduction three years ago. Too expensive, said I. Too slow. Requires too much memory. Why does one user need multiple tasks? Why is protected memory necessary on a single-user system? In retrospect, this seems to be a case of sour grapes: I could not afford to upgrade my personal computer to allow me to run OS/2.

I recently purchased a new machine with a 33-MHz 80386, a VGA monitor, and a fast 150-Mbyte hard drive. Everything looks very different from this new perspective. Sure, OS/2 has much more overhead than DOS, but at 33 MHz, you don't really notice. Sure, 4 Mbytes is a lot of memory, but the price of RAM has declined significantly. Multitasking really does make sense — never having to wait for the machine to finish one task before I can start another increases productivity. And, finally, while developing new software, protection from your own errant programs is a real advantage — it virtually eliminates having to reboot because of system crashes. So with that confession and recantation out of the way, let's talk project:

Kermit is a file-transfer protocol that allows diverse computers to communicate. In May 1989, my article "Kermit Meets Modula-2" (also here in *DDJ*) described the PCKermit protocol and my implementation in Modula-2. This article describes how I ported PCKermit over to the OS/2 Presentation Manager (PM).

Brian is an instructor of computer systems technology at the British Columbia Institute of Technology. He can be reached at 3700 Willingdon Ave., Burnaby, B.C., Canada V5G 3H2.

OS/2 Mini-Primer

OS/2 and PM are very large systems. The OS/2 kernel has over 250 functions; PM has over 500. This collection of functions is referred to as the Application Program Interface, or API. The Kernel API provides all of the traditional operating system services, including memory management, process control, and input/output. Using only the Kernel API, a programmer can write sophisticated character-based programs without ever having to resort to direct access to the hardware (as was often required under DOS).

The Kernel API is divided into four sections: *Dos*, *Kbd*, *Mou*, and *Vio*. The *Kbd*, *Mou*, and *Vio* provide fast, flexible access to the keyboard, mouse, and video systems, respectively, and are used mainly in traditional character-based applications. Presentation Manager takes control of the keyboard and mouse, so the *Kbd* and *Mou* group of functions is not used in a PM application. PM has many options available for controlling the screen; graphics support (both vector and raster) is available, as is support for various proportional and non-proportional fonts. The *Vio* group of functions, in a slightly altered form called "Advanced Vio" (*AVio*) is also available to a PM application. (My implementation of Kermit uses *AVio* functions to emulate a standard video display terminal (VDT) and thereby allows interaction with mainframe computers.) The DOS portion of the OS/2 kernel provides for disk I/O, task management, and memory management for both character-based and PM applications.

The Presentation Manager API is also divided into sections: *Dev*, *Gpi*, and *Win*. The functions in the *Dev* group are used to access PM device drivers, which allows for device-independent graphics. The *Gpi* group provides the drawing routines (lines, curves, bitmaps, fonts, and so on) for PM. The *Win* functions manage the PM windows including menus, scroll bars, and dialog boxes. Each *Win* function is quite "low level," which often makes for convoluted code (several functions, each with many parameters, must be called to accomplish some small task).

Besides the function libraries just described, PM makes use of resource files to store application-specific information about menus, dialog boxes, and so on. A resource file is a text file of source code that describes, for example, the form of a pull-down menu. The resources are compiled (with a resource compiler), and eventually linked to the application. The source code for many types of resources can be either developed manually or generated automatically using a dialog editor, for example.

On the Workbench

To develop programs for OS/2 and PM you will need a fairly extensive (and expensive) set of tools. Probably the most important "tool" is the documentation. The *Microsoft OS/2 Programmers's Reference* (volumes 1-3) is es-

(continued from page 70)
sential. There is also a fourth volume which covers OS/2 v1.2; that volume has remained on my shelf because I am running OS/2 v1.1. Because the Microsoft manuals are a bit light on examples, you will also want to get Charles Petzold's *Programming the OS/2 Presentation Manager* (Microsoft Press, 1989), and either Ray Duncan's *Advanced OS/2 Programming* (Microsoft Press, 1989) or Peter Norton and Robert LaFore's *Inside OS/2* (Brady Books, 1988) on the kernel. Gordon Letwin's *Inside OS/2* (Microsoft Press, 1988) is also a good overall OS/2 primer. For setting up and running (not programming) OS/2, you might consider *Using OS/2* by Halliday, Minasi, and Gobel (Que Books, 1989).

Of course, you will need a compiler or assembler that supports OS/2 and PM. Many of the Microsoft products obviously qualify. And other compilers are starting to support OS/2 and PM. Whatever language you choose to use, you will need at least the ability to read C code — all of the Microsoft documentation is written in C.

For this project, I used Stony Brook Modula-2, in part because my original implementation of Kermit was written in Modula-2 (albeit using the Logitech compiler), and in part because I prefer Modula-2 to C. The Stony Brook compiler proved an excellent choice; although much of the system was still in beta test, I found only one problem (which I was able to correct easily because I had the source code for the offending module).

Finally, you will need the resource set: Icon editor, dialog editor, font editor, resource compiler, and a few other miscellaneous tools. All of these tools are available from Microsoft as the OS/2 Softset. Alternately, you can purchase the OS/2 PM Software Development Kit which includes the Microsoft documentation, Petzold's book, and the tools (but no compiler). While purchasing the books and tools separately seems to be more economical, the SDK includes example code and useful programs (for example, PMCAP — a screen capture utility) that are not included with the Softset.

Format of a PM Program

Most GUIs (that is, GEM, Windows, Macintosh, and OS/2-PM) have a very similar structure:

1. The main program sits in a loop fetching messages from the operating system and dispatching those messages to a window procedure.
2. The window procedure intercepts the messages (usually with a very long *CASE* statement) and processes the messages, often by calling other functions.
3. Any other part of the program that wants to get anything done must communicate with the window procedure by posting messages (which the *main* fetches and dispatches: see steps 1 and 2).

Virtually everything is done with messages. For example, a PM program never calls a function such as *getchar( )* or *Read( )* to get a character from the keyboard. Instead, a program must continually look for a WM_CHAR message, and then translate the parameters that come along with the message to find out what key has been struck.

One tremendous advantage with OS/2 (compared to the other GUIs) is that preemptive multitasking allows independent threads to execute in the background and post messages to the main window (to let the user know what is going on). PCKermit makes use of multiple threads for terminal emulation (connecting to the host), as well as for sending and receiving files.

Down to Business

The main program module, PCKERMIT.MOD (see Listing One, page 109), begins on line 1. (Note that the listings are sequentially numbered for easy reference.) PCKermit initializes the window system and message queue, registers two window classes (for the main window and a child window — the child window is used during file transfer), creates and resizes the main window, and then sits in the message loop waiting for termination.

PCKermit is fairly traditional (for PM), except that after creating the main window, PCKermit immediately determines the size and position of the desktop (see line 99), records this in the global variable *Pos*, and then expands the window to nearly full size (line 100-102). The variable *Pos* is used again in the Shell module to ensure that the window is always correctly sized and positioned. The window is either sized to three device units smaller than the desktop (during file transfer), or is maximized (during terminal emulation). When the window is maximized or restored from maximum (by calling the *WinSetWindowPos* function again), an extra term must be included in the final parameter: Either *SWP_RESTORE* (line 543-544) or *SWP_MAXIMIZE* (line 552-554). A similar strategy is used to keep the child window properly sized (the child window is used for displaying status messages during file transfer).

SHELL.DEF (Listing Two, page 109) and SHELL.MOD (Listing Nine, Page 111) represent the most important module, as it is where all messages are processed (mostly in *WindowProc*, *DoMenu*, and *ChildWindowProc*). Several global variables and constants are defined here (that is, *Class* and *ClientWindow*). Besides the two window procedures, there are a number of ancillary procedures for controlling the windows: *SetFull*, *SetRestore*, and *SetMaximize* control the size of the window (PCKermit "insists" on a full window to properly emulate the TVI950 video screen); *Enable* and *Disable* "gray out" any menu item that is not currently accessible; *Check* and *Uncheck* indicate the currently selected video color scheme; *DoMenu* is called from *WindowProc* to process WM_COMMAND messages (these are mainly commands that result from the user interacting with menus); several dialog procedures (for example, *BaudDlgProc*) allow the user to make choices via pop-up dialog boxes (each dialog box is a window, and therefore needs its own window procedure); *KeyTranslate* processes keyboard messages and translates them into standard codes (for use by the Term module).

Preemptive multitasking allows independent threads to execute in the background, and post messages to the main window

The window procedure, *WindowProc* (line 1088), is called only by PM. Its purpose is to process messages (including standard system messages such as WM_CREATE, WM_INITMENU, WM_COMMAND, and WM_PAINT, as well as PCKermit messages such as WM_SETFUL, WM_SETRESTORE, and WM_TERM). Besides the message itself (which is really just an integer), the window procedure is passed two message parameters (*mp1* & *mp2*). The form of these two parameters depends upon the particular message. For example, in the case of the WM_TERM message, the message parameters are used to pass only a single character. In the case of the WM_CHAR message, the parameters contain a wide range of keyboard information (ASCII code, scan code, Control/Alt/Shift condition, time of keypress, and several flags). In some cases, the message parameters are not used at all.

One of the messages processed by *WindowProc*, WM_COMMAND, is passed on to *DoMenu* (line 743). This procedure processes all messages that result from the user clicking on a menu or using one of the accelerator keys. This user interaction often results in other child windows, called "dialog boxes," being created. When the dialog box is on the screen, another window procedure takes over. For example, if the user clicks on the Options menu, and then chooses baud rate . . . , a dialog box (with radio buttons and an OK push button) appears on the screen. While this dialog box is on the screen, the *BaudDlgProc* (line 863), takes over control of the window (until the user clicks on OK, or presses the Return or Esc key). Each of the dialog boxes or messages boxes (PCKermit has nine) has its own window procedure.

The child window procedure, *ChildWindowProc* (line 1203), is also called only by PM (as are all window procedures), and processes a variety of system and PCKermit messages. The *WM_PAD* and *PM_DL* messages are from the PAD (Packet Assembler Dis-

(continued from page 73)
assembler) and DataLink Modules, respectively, and are used mainly to allow these modules (which are running as independent threads) to keep the user informed: The message parameters indicate what message should be displayed on the screen.

When the user selects Send or Receive (either through the menus or via accelerator keys), a fairly complex chain of events is brought into play. In *DoMenu* at lines 776 or 780 the IDM_SEND or IDM_REC messages are recognized. In the case of IDM_SEND, a dialog box is invoked. The dialog box procedure *SendFNDlgProc* on line 983 allows the user to enter a filename; PM returns a pointer to the filename that the user enters.

In the case of either IDM_SEND or IDM_REC, the *MakeChild* procedure (line 630) is called next. This procedure first forces the main window to full size, then disables several menu items (because we don't want the user to try to change the baud rate halfway through a file transfer, line 639), next creates a standard window (line 649), sizes and positions the window (line 661), puts an appropriate message in the window (line 666), and finally makes the new child window the active window (line 668).

After the window is ready, we must set it up so that we can send messages. PCKermit uses the *AVio* for this, and lines 671–673 set up the *hvps* (Handle to *Vio* Presentation Space). Back in *DoMenu*, on lines 779 or 782, a new thread is created. OS/2 will then schedule that thread (which will either send or receive a file) on the next available time slice. The thread will terminate itself when file transfer is complete (or if five consecutive errors occur). Before the thread actually terminates itself (by calling *DosExit*), it sends messages to its window procedure to remove the window and restore the menus.

When the user selects Connect mode (either through menus or via keyboard accelerator), the IDM_CONNECT message is recognized (line 762). As in the case of Send and Receive, various menu items are disabled, and a presentation space is set up. Unlike Send and Receive, a new thread is not started, but an existing thread is resumed (line 774); the thread was set up during initialization (line 1145) and then immediately suspended (next line).

Next Time

If you've examined the listings up to this point, you'll notice that I haven't mentioned Listings Three through Eight. These are the definition (.DEF) files for the modules to be covered in Part II. Listing Three, (page 109) for example, is the .DEF file for the Term module which performs TVI950 terminal Emulation. Also note the definition of the *Dir* procedure (line 176) in Listing Three. Besides displaying a directory, this function allows the user to log onto a different drive, or to change to a different directory. This feature is necessary in a Kermit program, as only a file name must be specified when sending (you cannot send a file based upon a complete path). Part II examines these issues in greater detail.

Screen.DEF (Listing Four, page 109) gives you a flavor of the Screen module to be presented in Part II, which performs low-level screen and video I/O using OS/2's *VIO* functions. And CommPort.DEF (Listing Seven, page 111) presents a module originally supplied by Stony Brook that I've enhanced by adding extra buffering, for instance, to get around an OS/2 limitation. Of course, I'll also examine some of the unexpected problems associated with porting PCKermit to OS/2. Until next time. . . .

DDJ

(Listings begin on page 109.)

Vote for your favorite feature/article.
Circle Reader Service **No. 7.**

Listing One *(Text begins on page 16.)*

```
DEFINITION MODULE ModPlot;

(* Title   : High level Modula-2 Graphics library interface
Author  : Judy Auping
System  : PC Graphics
Compiler: LOGITECH MODULA-2/86
*)

  FROM DataDefs IMPORT
    DeviceType,UnitsType,SizeType,AngleType,LineType,ColorType,
    FontType,OriginType,SymbolType,ModeType,STRING80;

  EXPORT QUALIFIED
    GraphInit,SetPlotDevice,SetPlotArea,SetScaledArea,SetScale,SetUnits,
    SetCharSize,SetLabelAngle,SetLineType,SetPenColor,SetBackgroundColor,
    SetLabelOrigin,SetFixedDigits,ReturnRatio,SetFontType,
    Draw,Move,IncDraw,IncMove,DrawXAxis,DrawYAxis,DrawAxes,DrawGrid,
    DrawLabel,DrawLabelledAxes,DrawLabelledGrid,DrawFrame,DrawSymbol,
    Where,NewScreen,CloseGraphics;

  PROCEDURE GraphInit;
     (* Initializes system-MUST be called before any output generated *)
  PROCEDURE SetPlotDevice(GraphDevice: DeviceType);
     (* Selects output device for subsequent graphics commands *)
  PROCEDURE SetPlotArea(XMin,XMax,YMin,YMax: REAL);
     (* Sets the 'clip' area in % of absolute device boundaries*)
  PROCEDURE SetScaledArea(XMin,XMax,YMin,YMax: REAL);
     (* Sets the area in % of absolute device boundaries to which
        subsequent SetScale takes effect*)
  PROCEDURE SetScale(XMin,XMax,YMin,YMax: REAL);
     (* Sets the user scale *)
  PROCEDURE SetUnits(GraphUnits: UnitsType);
     (* Sets either User (Scaled units) or Device (absolute units) *)
  PROCEDURE SetCharSize(CharSize: SizeType);
     (* Sets character size for subsequent labels *)
  PROCEDURE SetLabelAngle(LabelRotation: AngleType);
     (* Sets the angle of rotation of subseqent labels *)
  PROCEDURE SetLineType(LineTypeSelected: LineType);
     (* Sets the line type for subsequent Draw commands *)
  PROCEDURE SetPenColor(PenColor: ColorType);
     (* Sets the pen color for subsequent output *)
  PROCEDURE SetBackgroundColor(BackgroundColor: ColorType);
     (* Sets the background color (screen only) *)
  PROCEDURE SetLabelOrigin(LabelOrigin: OriginType);
     (* Determines orientation relative to current position with
        which subsequent labels will be drawn *)
  PROCEDURE SetFixedDigits(XNumDigits,YNumDigits: CARDINAL);
     (* Sets number of digits to right of decimal point for
        subsequent DrawLabelledAxes or DrawLabelledGrid *)
  PROCEDURE ReturnRatio(): REAL;
     (* Returns the ratio of the physical dimensions of the
        plotting area of the current device *)
  PROCEDURE SetFontType(FontTypeSelected: FontType);
     (* Sets the font type for subsequent label commands *)
  PROCEDURE Draw(XCoord,YCoord: REAL);
     (* Draw a line from the active position to the specified coords *)
  PROCEDURE Move(XCoord,YCoord: REAL);
     (* Move the active position to the specified coordinates *)
  PROCEDURE IncDraw(XIncr,YIncr: REAL);
     (* Do an incremental draw from the active position *)
  PROCEDURE IncMove(XIncr,YIncr: REAL);
     (* Do an incremental move to the new active position *)

  (* In the following set of axis drawing and labelling procedures,
     the variables XIntercept, YIntercept, XTicSpacing, YTicSpacing
     XMin, XMax, YMin, and YMax are all interpreted according to the
     most recent SetUnits, SetScaledArea, and Set Scale commands.
     MajorCount is an integer that specifies the number of tic intervals
     between major tic marks.  In DrawLabelledAxes and DrawLabelledGrid,
     if MajorCount is positive, tic marks are drawn perpendicular to the
     corresponding axis; if negative, tic marks are parallel.
     MajorTicFrac specifies the size of the major tics as a percentage
     of the length of the corresponding axis.*)

  PROCEDURE DrawXAxis(YIntercept,TicSpacing,XMin,XMax: REAL;
                  MajorCount: INTEGER;
                  MajorTicFrac: REAL);
     (* Draw an X-axis from XMin to XMax at the specified y-intercept*)
  PROCEDURE DrawYAxis(XIntercept,TicSpacing,YMin,YMas: REAL;
                  MajorCount: INTEGER;
                  MajorTicFrac: REAL);
     (* Draw a Y-axis from YMin to YMax at the specified x-intercept*)
  PROCEDURE DrawAxes(XIntercept,YIntercept,XTicSpacing,YTicSpacing: REAL;
                XMajorCount,YMajorCount: INTEGER;
                XMajorTicFrac,YMajorTicFrac: REAL);
     (* Draws full-scale axes intersecting at XIntercept and YIntercept*)
  PROCEDURE DrawGrid(XIntercept,YIntercept,XTicSpacing,YTicSpacing: REAL;
                XMajorCount,YMajorCount: INTEGER;
                XMinorTicFrac,YMinorTicFrac: REAL);
     (* Draws a full-scale grid, with lines spaced symmetrically
        around XIntercept,YIntercept *)
  PROCEDURE DrawLabel(LabelString: ARRAY OF CHAR);
     (* Writes LabelString at the current active position according
        to the current CharSize, LabelOrigin, and LabelRotation settings*)
  PROCEDURE DrawLabelledAxes(XIntercept,YIntercept,XTicSpacing,
                          YTicSpacing: REAL;
                          XMajorCount,YMajorCount: INTEGER;
                          XMajorTicFrac,YMajorTicFrac: REAL);
     (* Draws a pair of axes in the same manner as DrawAxes.  Puts
        labels at the major tic marks according to the current
        CharSize, XNumDigits, and YNumDigits settings *)
  PROCEDURE DrawLabelledGrid(XIntercept,YIntercept,XTicSpacing,
                          YTicSpacing: REAL;
                          XGridSpacing,YGridSpacing: INTEGER;
                          XMajorTicFrac,YMajorTicFrac: REAL);
     (* Draws a full-scale grid as in DrawGrid and labels the grid
        lines as in DrawLabelledAxes *)
  PROCEDURE DrawFrame;
     (* Draws a box around the current plotting area *)
  PROCEDURE DrawSymbol(XCoord,YCoord: REAL;
                    Symbol: SymbolType;
                    Size: REAL);
     (* Draws the indicated symbol centered at XCoord,YCoord.
        Size is specified in mm *)
  PROCEDURE Where(VAR XCoord,YCoord: REAL);
     (* Returns the coordinate values of the current active position*)
  PROCEDURE NewScreen(Mode: ModeType);
     (* Clears the screen and sets the mode. No cursor in Menu mode*)
  PROCEDURE CloseGraphics;
     (* Cleans up the interrupts and restores to normal *)
END ModPlot.
```

End Listing One

Listing Two

```
DEFINITION MODULE GDriver;
(* Title   : Low Level CRT and Plotter Draw module
Author  : Chris Johnston
System  : Modula-2 Plotting System
Compiler: LOGITECH MODULA-2/86
*)

  FROM DataDefs IMPORT ModeType, STRING80, DevPresentType;

    EXPORT QUALIFIED
      ReadDevices, DrawAbs, MoveAbs, DrawString, SetMode, CleanUp;

   PROCEDURE ReadDevices(VAR DevicesPresent : DevPresentType);
     (* Finds out what devices are available: EGA/CGA and
         IBM7372A/IBM7372B/IBM7371. An HP 7470 is reported
         as an IBM7371 and the HP 7475 is reported as an
         IBM7372A or B depending upon the paper size selected *)

   PROCEDURE DrawAbs(XCoord, YCoord : CARDINAL);
     (* Draw a line from the current location to XCoord, YCoord
         on the selected device. Line type and color read from system
         globals  *)

   PROCEDURE MoveAbs(XCoord, YCoord : CARDINAL);
     (* Move from the current location to XCoord, YCoord on the
         selected device with the pen raised. *)

   PROCEDURE DrawString(XCoord, YCoord : CARDINAL;
                        LabelString    : ARRAY OF CHAR);
     (* Draw the character string LabelString starting at XCoord, YCoord.
         The font, color, size, and rotation are selected from system
         globals *)

   PROCEDURE SetMode( Mode : ModeType);
     (* Set the mode to text or graphics and clear the screen. This call has
         ** NO EFFECT ** if the selected device is a plotter *)

   PROCEDURE CleanUp;
     (* clean up the interrupt drivers and the character set at the end. *)

END GDriver.
```

End Listing Two

Listing Three

```
DEFINITION MODULE DataDefs;

(* Title   : Data Definitions
Author  : Judy Auping
System  : PC Graphics
Compiler: LOGITECH MODULA-2/86
*)

  EXPORT QUALIFIED
    DeviceType,UnitsType,SizeType,AngleType,LineType,ColorType,
    FontType,OriginType,SymbolType,DevPresentType,ModeType,STRING80,
    GraphDevice,LineTypeSelected,CharSize,FontTypeSelected,
    PenColor,BackgroundColor,LabelRotation,DeviceXMax,DeviceYMax,
    ErrorString, DriverError;

  TYPE
    DeviceType = (EGA,CGA,IBM7372A,IBM7372B,IBM7371);
    UnitsType = (User,Device);
    SizeType = (Small,Med,Large,XLarge);
    AngleType = (Deg0,Deg45,Deg90,Deg135,Deg180,Deg225,Deg270,Deg315);
    LineType = (Solid, EndPoint, Dotted, ShortDash, LongDash);
```

```
    ColorType = (Black,Blue,Green,Cyan,Red,Magenta,Brown,White,
                 DarkGray,LightBlue,LightGreen,LightCyan,
                 LightRed,LightMagenta,Yellow,IntensifiedWhite);
    FontType = (Standard,Italic);
    OriginType = (UpperRight,CenterRight,LowerRight,UpperMiddle,
                  CenterMiddle,LowerMiddle,UpperLeft,CenterLeft,
                  LowerLeft);
    SymbolType = (Circle,Square,Triangle,Asterisk,Cross,Plus);
    DevPresentType = ARRAY DeviceType OF BOOLEAN;
    ModeType = (Graphics,Text,Menu);
    STRING80 = ARRAY[0..79] OF CHAR;

  VAR
    GraphDevice: DeviceType;
    LineTypeSelected: LineType;
    CharSize: SizeType;
    FontTypeSelected: FontType;
    PenColor: ColorType;
    BackgroundColor: ColorType;
    LabelRotation: AngleType;
    DeviceXMax,DeviceYMax: CARDINAL;
    DriverError: BOOLEAN;
    ErrorString: STRING80;

END DataDefs.
```

End Listing Three

Listing Four

```
MODULE Example;
(* Title: Example of using the ModPlot graphics library
Author: Judy Auping
System: PC Graphics
*)

  FROM DataDefs IMPORT
    DeviceType,SizeType,ColorType,OriginType;

  FROM ModPlot IMPORT
    GraphInit,SetPlotArea,SetScaledArea,SetScale,DrawAxes,SetPenColor,
    Move,Draw,DrawFrame,NewScreen,CloseGraphics,SetLabelOrigin,DrawLabel,
    SetCharSize,IncDraw,SetPlotDevice;

  FROM MathLib0 IMPORT sin;

  FROM InOut IMPORT Read,WriteString,WriteLn;

TYPE
  CornerType = (UpLeft,UpRight,LowLeft,LowRight);

CONST
  NPnts = 1000; pi = 3.14159;

VAR
  XValue: ARRAY[1..NPnts] OF REAL;
  YValue: ARRAY [UpLeft..LowRight] OF ARRAY[1..NPnts] OF REAL;
  NumTerms: ARRAY[UpLeft..LowRight] OF CARDINAL;
  IPlot: CornerType;
  IPnt,ITerm: CARDINAL;
  x,NextTerm: REAL;
  Input: CHAR;

PROCEDURE GeneratePlotArrays;
(* This procedure generates arrays of data points for the Fourier series
   approximation of a sawtooth wave, where

       y = 2 (sin x - sin(2x)/2 + sin(3x)/3 - sin(4x)/4 + ... )

   The XValue array contains the x values for the plot in units of pi, where
   the values vary from zero to  4pi.

   The YValue array of arrays contains four arrays of y values for
   different numbers of terms in the summation approximation. *)

BEGIN
  WriteString("Generating approximation functions"); (* Inform user *)
  NumTerms[UpLeft] := 5;     NumTerms[UpRight] := 10;
  NumTerms[LowLeft] := 20;  NumTerms[LowRight] := 100;

  FOR IPnt := 1 TO NPnts DO

    IF (IPnt MOD 100)=0 THEN
      WriteString(" ."); (* Let the user know the progress of *)
    END (* if *);        (* the calculations.*)

    FOR IPlot := UpLeft TO LowRight DO
      YValue[IPlot,IPnt] := 0.0;  (* Initialize the terms. *)
    END (* for *);

    x := FLOAT(IPnt) * (4.0 * pi)/FLOAT(NPnts);
    XValue[IPnt] := x/pi;

    FOR ITerm := 1 TO NumTerms[LowRight] DO
      IF (ITerm MOD 2)=0 THEN (* even terms are negative *)
        NextTerm := -2.0 * sin(FLOAT(ITerm)* x)/FLOAT(ITerm);
      ELSE                    (* odd terms are positive,*)
        NextTerm := 2.0 * sin(FLOAT(ITerm)* x)/FLOAT(ITerm);
      END (* if *);

      FOR IPlot := UpLeft TO LowRight DO
        IF ITerm<=NumTerms[IPlot] THEN
          YValue[IPlot,IPnt] := YValue[IPlot,IPnt] + NextTerm;
        END (* if *);
      END (* for *);
    END (* for *);
  END (* for *);
END GeneratePlotArrays;
```

(continued on page 78)

Listing Four *(Listing continued, text begins on page 16.)*

```
BEGIN
  GeneratePlotArrays;
  GraphInit;
  SetPlotDevice(IBM7372A);
  WriteLn; WriteString("Drawing plot . . ."); (*Let user know where we are*)

  FOR IPlot := UpLeft TO LowRight DO  (*Draw a plot for each array*)
    CASE IPlot OF (* For each array, choose the appropriate plotting area*)
      UpLeft:SetPlotArea(0.0,45.0,60.0,100.0);  (*Upper left corner *)
           SetScaledArea(5.0,40.0,62.0,94.0);
     |UpRight: SetPlotArea(55.0,100.0,60.0,100.0); (*Upper right corner*)
           SetScaledArea(60.0,95.0,62.0,94.0);
     |LowLeft:SetPlotArea(0.0,45.0,0.0,40.0); (*Lower left*)
           SetScaledArea(5.0,40.0,2.0,34.0);
     |LowRight:SetPlotArea(55.0,100.0,0.0,40.0); (*Lower right*)
           SetScaledArea(60.0,95.0,2.0,34.0);
    END (* case *);

    SetScale(0.0,4.0,-4.0,4.0); (* remember, x is in units of pi *)
    SetPenColor(Black);
    DrawAxes(0.0,0.0,1.0,1.0,2,2,3.0,2.0); (* draw axes without labels *)
    (* Labels are drawn separately so we can put 'pi' on x-axis labels*)
    SetCharSize(Small); (* Label the axes *)
    SetLabelOrigin(CenterLeft); (* First, the y-axis *)
    Move(-0.05,4.0); DrawLabel("4");
    Move(-0.05,2.0); DrawLabel("2");
    Move(-0.05,0.0); DrawLabel("0");
    Move(-0.05,-2.0); DrawLabel("-2");
    Move(-0.05,-4.0); DrawLabel("-4");

    SetLabelOrigin(LowerMiddle); (* Then the x-axis *)
    Move(2.0,-0.5); DrawLabel("2pi");
    Move(4.0,-0.5); DrawLabel("4pi");

    CASE IPlot OF (*Set a new pen color for each plot *)
      UpLeft: SetPenColor(Red);
     |UpRight: SetPenColor(Green);
     |LowLeft: SetPenColor(Blue);
     |LowRight: SetPenColor(Magenta);
    END (* case *);

    Move(0.0,0.0); (* Start at the origin *)
    FOR IPnt := 1 TO NPnts DO
      Draw(XValue[IPnt],YValue[IPlot,IPnt]); (*Draw to each point*)
    END (* for *);

    SetPenColor(Black);
    DrawFrame; (*Draw a box around the plot for this array *)

    Move(0.75,4.7);
    SetLabelOrigin(CenterRight);
    SetCharSize(Small);
    CASE IPlot OF  (*Put the appropriate title on each plot*)
      UpLeft: DrawLabel("5 terms in series");
     |UpRight: DrawLabel("10 terms in series");
     |LowLeft:DrawLabel("20 terms in series");
     |LowRight:DrawLabel("100 terms in series");
    END (* case *);
  END (* for *);

  (*Now that all four plots have been drawn, put a title and
    the formula in the middle area on the page *)
  SetPlotArea(0.0,100.0,0.0,100.0); (* Set to full screen *);
  SetScaledArea(0.0,100.0,0.0,100.0);
  SetScale(0.0,100.0,0.0,100.0);

  Move(9.0,53.0);
  SetCharSize(Med);
  SetLabelOrigin(CenterRight);
  DrawLabel("FOURIER SERIES APPROXIMATION TO A SAWTOOTH WAVE");
  Move(17.0,47.0);
  SetCharSize(Small);
  DrawLabel("y = 2 {sin(x) - sin(2x)/2 + sin(3x)/3 - sin(4x)/4 + ... }");

  CloseGraphics; (* Clean up and restore the system *)
END Example.
```

End Listings

Listing One *(Text begins on page 26.)*

```
C
C    >*************************************************************
      PROGRAM GLOBE
C     *************************************************************
C     PROGRAM TO DRAW A GLOBE AT A USER SPECIFIED ANGLE ON A GRAPHICS
C     SURFACE. INPUTS ALSO INCLUDE LOCATION OF GRATING LOBES REFERENCED
C     TO LONGITUDE AND LATITUDE.
      AUTHOR: SCIENTIFIC CONCEPTS
C     ------------------------------------------------------------
      IMPLICIT NONE
C
C
      INTEGER*2         I               !LOOP COUNTER
      INTEGER*2         J               !LOOP COUNTER
      INTEGER*2         PMOVE           !PEN CONTROL MOVE COMMAND
      INTEGER*2         PDRAW           !PEN CONTROL DRAW COMMAND
      INTEGER*2         PENC            !PEN CONTROL: 2=DRAW,3=MOVE
      INTEGER*2         TLU             !TERMINAL LOGICAL UNIT NUMBER
      INTEGER*2         ROW             !TEXT ROW POSITION
      INTEGER*2         COLUMN          !TEXT COLUMN POSITION
      INTEGER*2         NUMLOBES        !NUMBER OF GRATING LOBES REQUESTED
C
      REAL*8            GRLOBEX(10)     !X LOCATION FOR GRATING LOBE
      REAL*8            GRLOBEY(10)     !Y LOCATION FOR GRATING LOBE
      REAL*8            XPOS            !HORIZONTAL PIXEL POSITION
      REAL*8            YPOS            !VERTICAL PIXEL POSITION
      REAL*8            HORIZONTAL      !CALCULATED HORIZONTAL PIXEL POSITION
      REAL*8            VERTICAL        !CALCULATED VERTICAL PIXEL POSITION
      REAL*8            RADIUS          !RADIUS OF GLOBE CIRCLE
      REAL*8            TILT            !TILT ANGLE FOR GLOBE
      REAL*8            PI              !PI CONSTANT
      REAL*8            COSCONVER       !COS CONVERSION OF TILT IN RADIANS
      REAL*8            SINCONVER       !SIN CONVERSION OF TILT IN RADIANS
      REAL*8            ELEVATION       !CALCULATED LONGITUDE POSITION
      REAL*8            AZIMUTH         !CALCULATED LATITUDE POSITION
      REAL*8            GLOBEINC        !GRATING LOBE INCREMENT (RADIANS)
C
      CHARACTER         STEMP*8         !TEMPORARY STRING
C
C
      PARAMETER         (PMOVE=3,PDRAW=2)
C
      TLU=6
      NUMLOBES=0
      PI=3.14159265
C
C
C     HORIZONTAL,VERTICAL ARE COORDINATES OF ORIGIN
C
      WRITE(TLU,*)'ENTER ORIGIN COORDINATES (TRY 300,200 FOR EGA/VGA)'
      READ(TLU,*)HORIZONTAL,VERTICAL
C
      WRITE(TLU,*)'ENTER RADIUS OF CIRCLE (TRY 160 FOR EGA/VGA)'
      READ(TLU,*)RADIUS
C
      WRITE(TLU,*)'ENTER TILT ANGLE IN DEGREES (TRY 30)'
      READ(TLU,*)TILT
C
      WRITE(TLU,*)'HOW MANY GRATING LOBES (MAXIMUM=10) ? '
      READ(TLU,*)NUMLOBES
C
      IF (NUMLOBES.GT.10) THEN
        WRITE(TLU,*)' ERROR: TOO MANY GRATING LOBES REQUESTED!'
        STOP
      ELSE IF (NUMLOBES.GT.0) THEN
        DO I=1,NUMLOBES
          WRITE(TLU,*)'ENTER (X,Y) COORDINATES FOR POINT ',I
          READ(TLU,*)GRLOBEX(I),GRLOBEY(I)
        END DO
      ENDIF
C
C     INITIALIZE IBM PC TO MAXIMUM RESOLUTION ...
C
      CALL GINIT(TLU)
C
C     DRAW '+' AT ORIGIN
C
      XPOS=HORIZONTAL-4.5
      CALL PLOT(XPOS,VERTICAL,PMOVE)
      XPOS=HORIZONTAL+4.5
      CALL PLOT(XPOS,VERTICAL,PDRAW)
      YPOS=VERTICAL-3.6
      CALL PLOT(HORIZONTAL,YPOS,PMOVE)
      YPOS=VERTICAL+3.9
      CALL PLOT(HORIZONTAL,YPOS,PDRAW)
C
C     LABEL FIGURE WITH PARAMETERS
C
      ROW=24
      COLUMN=26
      WRITE(STEMP,'(F6.2)')TILT
      CALL TEXTLABEL(ROW,COLUMN,'TILT ANGLE (DEGREES)='//STEMP)
C
C     DRAW OUTER CIRCLE
C
      CALL PLOT(HORIZONTAL+RADIUS,VERTICAL,PMOVE)
      DO I=1,100
        XPOS=HORIZONTAL+RADIUS*COS(I*2*PI/100)
        YPOS=VERTICAL+RADIUS*SIN(I*2*PI/100)
        CALL PLOT(XPOS,YPOS,PDRAW)
      END DO
C
C     DRAW LATITUDES
C
      TILT=TILT*PI/180.0
      COSCONVER=COS(TILT)
      SINCONVER=SIN(TILT)
C
      DO I=1,12
        ELEVATION=PI/2-PI/12*I
        XPOS=HORIZONTAL
        YPOS=VERTICAL+RADIUS*(SIN(ELEVATION)*COSCONVER
     +       -COS(ELEVATION)*SINCONVER)
        CALL PLOT(XPOS,YPOS,PMOVE)
        PENC=2
        DO J=1,100
          AZIMUTH=J*2*PI/100.0
          IF (SIN(ELEVATION)*SINCONVER+COS(ELEVATION)*
     +        COS(AZIMUTH)*COSCONVER.GE.0.) THEN
            XPOS=HORIZONTAL+RADIUS*COS(ELEVATION)*SIN(AZIMUTH)
            YPOS=VERTICAL+RADIUS*(SIN(ELEVATION)*COSCONVER
     +           -COS(ELEVATION)*COS(AZIMUTH)*SINCONVER)
            CALL PLOT(XPOS,YPOS,PENC)
            PENC=2
          ELSE
            PENC=3
          END IF
        END DO
      END DO
C
C     DRAW LONGITUDES
C
      DO I=1,12
        AZIMUTH=I*PI/12
        YPOS=VERTICAL+RADIUS*COSCONVER
        CALL PLOT(HORIZONTAL,YPOS,PMOVE)
        PENC=2
        DO J=1,100
          ELEVATION=PI/2-J*2*PI/100
          IF (SIN(ELEVATION)*SINCONVER+COS(ELEVATION)*
     +        COS(AZIMUTH)*COSCONVER.GE.0.) THEN
            XPOS=HORIZONTAL+RADIUS*COS(ELEVATION)*SIN(AZIMUTH)
            YPOS=VERTICAL+RADIUS*(SIN(ELEVATION)*COSCONVER
     +           -COS(ELEVATION)*COS(AZIMUTH)*SINCONVER)
            CALL PLOT(XPOS,YPOS,PENC)
            PENC=2
          ELSE
            PENC=3
          END IF
        END DO
      END DO
C
C
C     DRAW GRATING LOBES
C
      IF (NUMLOBES.GT.0) THEN
        DO I=1,NUMLOBES
          XPOS=HORIZONTAL+GRLOBEX(I)+RADIUS
          YPOS=VERTICAL+GRLOBEY(I)
          CALL PLOT(XPOS,YPOS,PMOVE)
C
          DO J=1,100
            GLOBEINC=J*PI/50
            XPOS=HORIZONTAL+GRLOBEX(I)+RADIUS*COS(GLOBEINC+.04)
            YPOS=VERTICAL+GRLOBEY(I)+RADIUS*SIN(GLOBEINC+.04)
            IF((GRLOBEX(I)+RADIUS*COS(GLOBEINC))**2+
     +         (GRLOBEY(I)+RADIUS*SIN(GLOBEINC))**2.LT.RADIUS**2) THEN
              CALL PLOT(XPOS,YPOS,PDRAW)
            ELSE
              CALL PLOT(XPOS,YPOS,PMOVE)
            END IF
          END DO
        END DO
      END IF
C
C
C     PREPARE TO EXIT GRAPHICS AND RETURN TO NORMAL VIDEO ...
C
      CALL EXITGRAPHICS(TLU)
C
      END
C
C
      INCLUDE 'FGRAPH.FI'
C
C
C    >*************************************************************
      SUBROUTINE TEXTLABEL(ROW,COLUMN,STRING)
C     *************************************************************
C     SUBROUTINE TO WAIT FOR USER SIGNAL AND EXIT GRAPHICS MODE. TERMINAL
C     IS RESTORED TO PRE-VIDEO CONDITIONS...
C     ------------------------------------------------------------
      IMPLICIT NONE
C
      INCLUDE 'FGRAPH.FD'
C
      INTEGER*2         ROW             !TEXT ROW POSITION
      INTEGER*2         COLUMN          !TEXT COLUMN POSITION
C
      CHARACTER         STRING*(*)      !TEXT STRING FOR LABEL
C
      RECORD /RCCOORD/ CURPOS
C
C
C     OUTPUT USER SUPLIED STRING AT ROW,COLUMN ...
C
      CALL SETTEXTPOSITION(ROW,COLUMN,CURPOS)
      CALL OUTTEXT(STRING)
C
      RETURN
      END
C
C
C    >*************************************************************
      SUBROUTINE EXITGRAPHICS(TLU)
C     *************************************************************
C     SUBROUTINE TO WAIT FOR USER SIGNAL AND EXIT GRAPHICS MODE. TERMINAL
C     IS RESTORED TO PRE-VIDEO CONDITIONS...
C     ------------------------------------------------------------
      IMPLICIT NONE
```

(continued on page 82)

Listing One *(Listing continued, text begins on page 26.)*

```
C
      INCLUDE 'FGRAPH.FD'
C
      INTEGER*2         TLU             !TERMINAL LOGICAL UNIT NUMBER
      INTEGER*2         DUMMY           !DUMMY FUNCTION ARGUMENT
      INTEGER*2         ROW             !TEXT ROW POSITION
      INTEGER*2         COLUMN          !TEXT COLUMN POSITION
C
      ROW=25
      COLUMN=28
C
C
C     OUTPUT PROMPT AND WAIT FOR ENTER KEY ...
C
      CALL TEXTLABEL(ROW,COLUMN,'PRESS ENTER TO CONTINUE')
      READ(TLU,*)
C
C     RESET VIDEO MODE AND STOP
C
      DUMMY=SETVIDEOMODE($DEFAULTMODE)
C
      RETURN
      END
C
C
C    >*************************************************************
      SUBROUTINE GINIT(TLU)
C     *************************************************************
C     SUBROUTINE TO INITIALIZE IBM PC GRAPHICS MODE TO MAXIMUM
C     AVAILABLE RESOLUTION ...
C     -------------------------------------------------------------
      IMPLICIT NONE
C
      INCLUDE 'FGRAPH.FD'
C
      INTEGER*2         ERRC            !ERROR CODE RETURNED
      INTEGER*2         TLU             !TERMINAL LOGICAL UNIT NUMBER
      INTEGER*2         DUMMY           !DUMMY FUNCTION ARGUMENT
C
      LOGICAL*2         WINDINVERT      !INVERT WINDOW COORDINATES IF TRUE
C
      REAL*8            LOWERX          !LOWER X AXIS CORNER OF WINDOW
      REAL*8            LOWERY          !LOWER Y AXIS CORNER OF WINDOW
      REAL*8            UPPERX          !UPPER X AXIS CORNER OF WINDOW
      REAL*8            UPPERY          !UPPER Y AXIS CORNER OF WINDOW
C
C
C
C     INITIALIZE VIDEO MODE TO MAXIMUM RESOLUTION AVAILABLE
C
      ERRC=SETVIDEOMODE($MAXRESMODE)
      IF (ERRC.EQ.0) THEN
        WRITE(TLU,*)' ERROR: CANNOT SET VIDEO MODE'
        STOP
      END IF
C
      LOWERX=-3.0
      LOWERY=3.0
      UPPERX=-3.0
      UPPERY=3.0
      WINDINVERT=.TRUE.
      DUMMY=SETWINDOW(WINDINVERT,LOWERX,LOWERY,UPPERX,UPPERY)
C
      RETURN
      END
C
C
C    >*************************************************************
      SUBROUTINE PLOT(XPOS,YPOS,PENC)
C     *************************************************************
C     SUBROUTINE TO DRAW OR MOVE TO THE USER SPECIFIED POSITION 'XPOS,
C     YPOS' WITH PEN CONTROL AS DESIGNATED BY 'PENC'.
C     -------------------------------------------------------------
      IMPLICIT NONE
C
      INCLUDE 'FGRAPH.FD'
C
      INTEGER*2         DUMMY           !DUMMY FUNCTION ARGUMENT
      INTEGER*2         PENC            !PEN CONTROL: 2=DRAW,3=MOVE
C
      REAL*8            XPOS            !HORIZONTAL PIXEL POSITION
      REAL*8            YPOS            !VERTICAL PIXEL POSITION
C
      RECORD /WXYCOORD/ XY
C
      IF (PENC.EQ.2) THEN
        DUMMY=LINETO_W(XPOS,YPOS)
      ELSE
        CALL MOVETO_W(XPOS,YPOS,XY)
      END IF
C
      RETURN
      END
```

End Listing One

Listing Two

Top Level Fragment

```
C    >*************************************************************
      PROGRAM GLOBE
C     *************************************************************
C
C PROGRAM TO DRAW A GLOBE AT A USER SPECIFIED ANGLE ON A GRAPHICS
C SURFACE. INPUTS ALSO INCLUDE LOCATION OF GRATING LOBES REFERENCED
C TO LONGITUDE AND LATITUDE.
C AUTHOR: SCIENTIFIC CONCEPTS
C
C-------------------------------------------------------------------
.
.
.
      CALL GINIT          !INITIALIZE GRAPHICS DEVICE
.
.
.
      END
```

Layer 3: Graphics Primitives

```
C*****************************************************C
      SUBROUTINE GINIT
C*****************************************************C
C     PURPOSE: INITIALIZE GRAPHICS DEVICE CURRENTLY
C              SET BY GLOBAL VARIABLE 'DEVICETYPE' ...
.
.
.
      IF (DEVICETYPE.EQ.HPGL) THEN          !HP GRAPHICS DEVICE
        CALL HPGLINIT
      ELSE IF (DEVICETYPE.EQ.IBMPC) THEN    !IBM MODES CGA-VGA
        CALL IBMPCINIT
      ELSE IF (DEVICETYPE.EQ.TEK) THEN      !TEKTRONIX DEVICES
        CALL TEKINIT
      ELSE IF (DEVICETYPE.EQ.DECVT) THEN    !DEC VT340
        CALL DECVTINIT
      ELSE IF (DEVICETYPE.EQ.VAXSTA) THEN   !DEC VAXSTATION 2000
        CALL VAXSTAINIT
.
.
.     ELSE
        CALL INITERROR
      END IF
```

Layer 2: Graphics Device Drivers

```
C*****************************************************C
      SUBROUTINE IBMPCINIT
C*****************************************************C
C     PURPOSE: INITIALIZE CURRENT IBM PC GRAPHICS MODE
C              COLORS, RESOLUTION ETC ...
.
.
.
C
      IF (IBMMODE.EQ.EGACOLOR) THEN
        DUMMY=SETVIDEOMODE($ERESCOLOR)
      ELSE IF (IBMMODE.EQ.HERCULES) THEN
        DUMMY=SETVIDEOMODE($HERCMONO)
.
.
.
      END IF
C
      RETURN
      END
C
C*****************************************************C
      SUBROUTINE VAXSTAINIT
C*****************************************************C
C     PURPOSE: INITIALIZE VAXSTATION 200 GRAPHICS DEVICE
C              MODE, VIEWPORT ...
.
.
.
C
      LOWLX=1.0             !LOWER LEFT X COORDINATE
      LOWLY=1.0             !LOWER LEFT Y COORDINATE
      UPPRX=20.0            !UPPER RIGHT X COORDINATE
      UPPRY=20.0            !UPPER RIGHT Y COORDINATE
      DISPWIDTH=20.0
      DISPHEIGHT=20.0
C
     VD_ID=UIS$CREATE_DISPLAY(LOWLX,LOWLY,UPPRX,UPPRY,
     +                         DISPWIDTH,DISPHEIGHT)
     WD_ID=UIS$CREATE_WINDOW(VD_ID,'SYS$WORKSTATION')
C
.
.
.
      RETURN
      END
C
C
```

End Listings

Listing One *(Text begins on page 36.)*

```
UNIT Shapes;

INTERFACE

    USES Graph, Crt;

    TYPE
        {---------- Shape class ----------}
        Shape = OBJECT
            Color : BYTE;

            PROCEDURE Draw; VIRTUAL;
            END;

        ShapePtr = ^Shape;

        {---------- Box class ----------}
        Box = OBJECT (Shape)
            LowerRightX, LowerRightY : INTEGER;
            UpperLeftX,  UpperLeftY  : INTEGER;

            CONSTRUCTOR Create(ulx, uly, lrx, lry, c : INTEGER);
            PROCEDURE Draw; VIRTUAL;
            END;

        {---------- Ring class ----------}
        Ring = OBJECT (Shape)
            Xcenter, Ycenter : INTEGER;
            Radius : INTEGER;

            CONSTRUCTOR Create(x, y, rad, c : INTEGER);
            PROCEDURE Draw; VIRTUAL;
            END;


IMPLEMENTATION

    {---------- Shape methods ----------}

        PROCEDURE Shape.Draw;
        BEGIN
                { Does nothing; it's a virtual method place-holder }
        END;


    {---------- Box methods ----------}

    CONSTRUCTOR Box.Create(ulx, uly, lrx, lry, c : INTEGER);
        BEGIN
         UpperLeftX  := ulx;
         UpperLeftY  := uly;
         LowerRightX := lrx;
         LowerRightY := lry;
         Color       := c;
        END;

        PROCEDURE Box.Draw;
        BEGIN
                SetColor(Color);

                Rectangle(UpperLeftX,  UpperLeftY, LowerRightX, LowerRightY);
        END;

    {---------- Ring methods ----------}

        CONSTRUCTOR Ring.Create(x, y, rad, c : INTEGER);
        BEGIN
                Xcenter := x;
                Ycenter := y;
                Radius  := rad;
        Color   := c;
        END;

        PROCEDURE Ring.Draw;
        BEGIN
                SetColor(Color);

                Circle(Xcenter, Ycenter, Radius);
        END;

END. { UNIT Shapes }
```

End Listing One

Listing Two

```
PROGRAM Program1;

USES Graph, Crt, Shapes;

VAR
    R : Ring;
    B : Box;

    GrDriver, GrMode, GrResult : INTEGER;

    ch : CHAR;
PROCEDURE DrawShape(s : ShapePtr);
BEGIN
    s^.Draw;
END;

BEGIN
    R.Create(100,100,90,2);
    B.Create(25,10,175,175,4);

    GrDriver := DETECT;
    InitGraph(GrDriver,GrMode,'d:\tp');

    GrResult := GraphResult;

    IF GrResult <> GrOK THEN
        BEGIN
        WriteLn('Unable to initialize BGI: error ',grResult);
        Halt(1);
        END;

    DrawShape(@R);
    DrawShape(@B);

        REPEAT UNTIL KeyPressed;

        Ch := ReadKey;

    CloseGraph;

END.
```

End Listing Two

Listing Three

```
PROGRAM Program2;

USES Graph, Crt, Shapes;

VAR
    R1, R2 : Ring;
    B1, B2 : Box;

    F : FILE;

    GrDriver, GrMode, GrResult : INTEGER;

    ch : CHAR;

BEGIN
    R1.Create(100,100,90,2);
    R2.Create(0,0,0,0);
    B1.Create(25,10,175,150,4);
    B2.Create(0,0,0,0,0);

    GrDriver := DETECT;
    InitGraph(GrDriver,GrMode,'d:\tp');

    GrResult := GraphResult;

    IF GrResult <> GrOK THEN
        BEGIN
        WriteLn('Unable to initialize BGI: error ',grResult);
        Halt(1);
        END;

    Assign(F,'shapes.fil');
    Rewrite(F,1);

    BlockWrite(F,R1,SizeOf(R1));
    BlockWrite(F,B1,SizeOf(B1));

    Close(F);

    Assign(F,'shapes.fil');
    Reset(F,1);

    BlockRead(F,R2,SizeOf(R2));
    BlockRead(F,B2,SizeOf(B2));

    R2.Draw;
    B2.Draw;

        REPEAT UNTIL KeyPressed;

        Ch := ReadKey;

    Close(F);

    CloseGraph;

END.
```

End Listing Three

Listing Four

```
UNIT Shapes;

INTERFACE

    USES Graph, Crt;

    TYPE
       {---------- Shape class ----------}
       Shape = OBJECT
           Color : BYTE;

           PROCEDURE Draw;   VIRTUAL;

           PROCEDURE FWrite(VAR f : FILE); VIRTUAL;
           PROCEDURE FRead(VAR f : FILE);  VIRTUAL;
           END;

       ShapePtr = ^Shape;

       {---------- Box class ----------}
       Box = OBJECT (Shape)
           UpperLeftX,  UpperLeftY  : INTEGER;
           LowerRightX, LowerRightY : INTEGER;

           CONSTRUCTOR Create(ulx, uly, lrx, lry, c : INTEGER);
           PROCEDURE Draw; VIRTUAL;
```

(continued on page 86)

Listing Four *(Listing continued, text begins on page 36.)*

```
          PROCEDURE FWrite(VAR f : FILE); VIRTUAL;
          PROCEDURE FRead(VAR f : FILE);  VIRTUAL;
          END;

      {---------- Ring class ----------}
      Ring = OBJECT (Shape)
          Xcenter, Ycenter : INTEGER;
          Radius : INTEGER;

          CONSTRUCTOR Create(x, y, rad, c : INTEGER);
          PROCEDURE Draw; VIRTUAL;

          PROCEDURE FWrite(VAR f : FILE); VIRTUAL;
          PROCEDURE FRead(VAR f : FILE);  VIRTUAL;
          END;


IMPLEMENTATION

    {---------- Shape methods ----------}

        PROCEDURE Shape.Draw;
        BEGIN
    END;

    PROCEDURE Shape.FWrite(VAR f : FILE);
    BEGIN
    END;

    PROCEDURE Shape.FRead(VAR f : FILE);
    BEGIN
    END;


    {---------- Box methods ----------}

    CONSTRUCTOR Box.Create(ulx, uly, lrx, lry, c : INTEGER);
        BEGIN
         UpperLeftX  := ulx;
         UpperLeftY  := uly;
         LowerRightX := lrx;
         LowerRightY := lry;
         Color       := c;
        END;

        PROCEDURE Box.Draw;
        BEGIN
                SetColor(Color);

                Rectangle(UpperLeftX,  UpperLeftY, LowerRightX, LowerRightY);
        END;

    PROCEDURE Box.FWrite(VAR f : FILE);
    BEGIN
        BlockWrite(f,Color,      SizeOf(INTEGER));
        BlockWrite(f,UpperLeftX, SizeOf(INTEGER));
        BlockWrite(f,UpperLeftY, SizeOf(INTEGER));
        BlockWrite(f,LowerRightX,SizeOf(INTEGER));
        BlockWrite(f,LowerRightY,SizeOf(INTEGER));
    END;

    PROCEDURE Box.FRead(VAR f : FILE);
    BEGIN
        BlockRead(f,Color,      SizeOf(INTEGER));
        BlockRead(f,UpperLeftX, SizeOf(INTEGER));
        BlockRead(f,UpperLeftY, SizeOf(INTEGER));
        BlockRead(f,LowerRightX,SizeOf(INTEGER));
        BlockRead(f,LowerRightY,SizeOf(INTEGER));
    END;

    {---------- Ring methods ----------}

    CONSTRUCTOR Ring.Create(x, y, rad, c : INTEGER);
        BEGIN
                Xcenter := x;
                Ycenter := y;
                Radius  := rad;
        Color   := c;
        END;

        PROCEDURE Ring.Draw;
        BEGIN
                SetColor(Color);

                Circle(Xcenter, Ycenter, Radius);
        END;

    PROCEDURE Ring.FWrite(VAR f : FILE);
    BEGIN
        BlockWrite(f,Color,   SizeOf(INTEGER));
        BlockWrite(f,Xcenter, SizeOf(INTEGER));
        BlockWrite(f,Ycenter, SizeOf(INTEGER));
        BlockWrite(f,Radius,  SizeOf(INTEGER));
    END;

    PROCEDURE Ring.FRead(VAR f : FILE);
    BEGIN
        BlockRead(f,Color,   SizeOf(INTEGER));
        BlockRead(f,Xcenter, SizeOf(INTEGER));
        BlockRead(f,Ycenter, SizeOf(INTEGER));
        BlockRead(f,Radius,  SizeOf(INTEGER));
    END;

END. { UNIT Shapes }
```

End Listing Four

Listing Five

```
PROGRAM Program3;

USES Graph, Crt, Shapes;

VAR
    R1, R2 : Ring;
    B1, B2 : Box;

    F : FILE;

    GrDriver, GrMode, GrResult : INTEGER;

    ch : CHAR;

BEGIN
    R1.Create(100,100,90,2);
    R2.Create(0,0,0,0);
    B1.Create(25,10,175,150,4);
    B2.Create(0,0,0,0,0);

    GrDriver := DETECT;
    InitGraph(GrDriver,GrMode,'d:\tp');

    GrResult := GraphResult;

    IF GrResult <> GrOK THEN
        BEGIN
        WriteLn('Unable to initialize BGI: error ',grResult);
        Halt(1);
        END;

    Assign(F,'shapes.fil');
    Rewrite(F,1);

    R1.FWrite(F);
    B1.FWrite(F);

    Close(F);

    Assign(F,'shapes.fil');
    Reset(F,1);

    R2.FRead(F);
    B2.FRead(F);

    R2.Draw;
    B2.Draw;

        REPEAT UNTIL KeyPressed;

        Ch := ReadKey;

    Close(F);

    CloseGraph;

END.
```

End Listings

Listing One *(Text begins on page 42.)*

```
FUNCTION FASTSRCH% (BuffFile%, RecLen%, KeyLoc%, KeyVal$, StartRec%,
                                                    LastRec%, FoundRecs%())

    CONST physbufflen% = 10000

    DIM FarBuff(Dum%) AS STRING * physbufflen  '* string buffer in the far heap
    MaxFound% = UBOUND(FoundRecs%)  '* can only find as many as array will hold
    NumMatches% = 0                 '* initialize the number of matches found

    IF KeyLoc% = RecLen% THEN         '* if key is the last byte in the record:
     CompByte% = 0 : LastByte% = -1   '* modulo position to check will be 0
    ELSE                              '* and when calculating record number
     CompByte%=KeyLoc% : LastByte%=0  '* of the matching record,
                                      '* 1 must be subtracted
    END IF

    RecsPerRead% = physbufflen \ RecLen%
                     '* calc. the # of records scanned with each disk read
    BuffLen% = RecsPerRead% * RecLen%
                     '* the actual number of bytes in the buffer being used
    CurRec% = StartRec%  '* initialize CurRec

    DO                   '* OUTER LOOP *  This loop executes for each disk read
       BuffByte& = ((CurRec% - 1&) * RecLen%) + 1  '* first byte in next buffer
       GET BuffFile%, BuffByte&, FarBuff(0)     '* perform the actual disk i/o
       Match% = 0                               '* initialize match to false
       IF CurRec% + RecsPerRead% - 1 > LastRec% THEN     '* when the last disk
          BuffLen% = (LastRec% - CurRec% + 1) * RecLen% '* read goes past the
       END IF                                           '* number of records
                                                        '* to search, shorten
                                                        '* the number of bytes
                                                        '* in buffer to scan

       DO        '* MIDDLE LOOP * Loops for each match found in this buffer
           DO    '* INNER LOOP * Loops for each "false" match
               Match% = INSTR(Match% + 1, FarBuff(0), KeyVal$)
               Match% > BuffLen% THEN Match% = 0
                          '* if match was past valid data, ignore the match
               IF Match% = 0 THEN EXIT DO     '* no more matches in this buffer
           LOOP WHILE Match% MOD RecLen% <> CompByte%
                     '* keep searching if the match was in the wrong position
           IF Match% > 0 AND NumMatches% < MaxFound% THEN
                                    '* if a match and the array isn't full
               NumMatches% = NumMatches% + 1
                                    '* add into the array of found records
               FoundRecs%(NumMatches%) = _
                  ((Match% + LastByte%) \ RecLen%) + CurRec%
                                   '* CALCULATE THE ACTUAL RECORD NUMBER *
           END IF
       LOOP UNTIL Match% = 0      '* no more matches in this buffer, exit loop
        CurRec% = CurRec% + RecsPerRead%  '* calc. first record of next buffer
    LOOP UNTIL CurRec% > LastRec%     '* if 1st rec. of next buffer is > number
                                      '* of recs to search then were finished
    FASTSRCH% = NumMatches%           '* set return value = number of matches

END FUNCTION
```

End Listing One

Listing Two

```
DECLARE FUNCTION InstrASM% (BYVAL BuffSeg%, BYVAL BuffOff%, BYVAL BuffLen%,_
              BYVAL KeySeg%, BYVAL KeyOff%, BYVAL KeyLen%, BYVAL RecLen%)

FUNCTION InstrR% (StRec%, Buff() AS STRING * 10000, KeyVal$, RecLen%,
                                                          KeyPos%) STATIC

    KeyLen% = LEN(KeyVal$)                        '* Get the length of the key

    StartingOffs% = ((StRec% - 1) * RecLen%) + KeyPos% - 1
                                                  '* Calculate the 1st byte to search
    BuffLen% = LEN(Buff(0)) - StartingOffs%  '* Calculate # of bytes to search
    BuffSeg% = SSEG(Buff(0))                 '* Locate the buffer's segment
    BuffOff% = SADD(Buff(0)) + StartingOffs% '* and offset.

    KeySeg% = SSEG(KeyVal$)                   '* Locate the key's segment
    KeyOff% = SADD(KeyVal$)                   '* and offset.

    FoundPos% = InstrASM%(BuffSeg%, BuffOff%, BuffLen%,
                      KeySeg%, KeyOff%, KeyLen%, RecLen%) '* Perform a search

    IF FoundPos% = 0 THEN             '* If no match,
        InstrR% = 0                   '* set the record number to 0
    ELSE                              '* otherwise calculate the record number
        InstrR% = (FoundPos% + StartingOffs% + RecLen% - 1) \ RecLen%
    END IF

END FUNCTION
```

End Listing Two

Listing Three

```
.MODEL MEDIUM, BASIC
.CODE
INSTRASM  PROC  FAR  USES DS SI DI, BuffSeg,BuffStartAdd,BuffLen,KeySeg,\
                                      KeyAddr,KeyLen,RecLen
        PUSHF
        MOV     BX,BuffSeg      ; Move the Buffer Segment
        MOV     ES, BX          ;   into the EXTRA SEGMENT REGISTER
        MOV     BX,KeySeg       ; Move the Key String Segment
        MOV     DS,BX           ;   into the DATA SEGMENT REGISTER
        MOV     AX,RecLen       ; Move the Record Length to AX
        MOV     DX,BuffLen      ; Move the Length of the Buffer to DX
        MOV     BX,BuffStartAdd ; Move Buffer Starting Address to BX
        ADD     DX,BX           ; Add Buffer Offset(BX) to Buffer Length(DX)
        SUB     DX,1            ;   and subtract 1 for End of Buffer Address
        CLD                     ; Clear the direction flag

FinishedYet:
        CMP     BX,DX           ; Next Record Address>End Of Buffer Address?
        JA      NoMoreBuff      ;   if so, quit.

        MOV     DI,BX           ; Move Next Record Address to Dest. Index
        MOV     SI,KeyAddr      ; Move Key Address to Source Index
        MOV     CX,KeyLen       ; Move Key Length to CX as REPE counter

        REPE CMPSB              ; Compare strings and if equal
        JE      Match           ;   go to Match (* can use JA or JB to alter *)

        ADD     BX,AX           ; Add Record Length(AX) to Next Record Address
        JMP     FinishedYet     ; Loop back and continue checking for a match

NoMoreBuff:
        XOR     AX,AX           ; Set AX to 0 for return in function value
        JMP     TheEnd          ; Go to end

Match:
        MOV     AX,BX           ; Move Current Record Offset to AX for return
        SUB     AX,BuffStartAdd ;   Subtract Starting Address of Buffer and
        ADD     AX,1            ;   add 1 to get actual offset w/i Buffer

TheEnd:
        POPF
        RET

INSTRASM ENDP
        END
```

End Listings

Listing One *(Text begins on page 50.)*

```
/* 68A.C - Main driver for generic assembler. */

#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <time.h>
#include <dir.h>

#include "68defs.h"
#include "68err.h"
#include "68parse.h"
#include "68list.h"
#include "68assem.h"
#include "68symtab.h"

void assembler_print_errors ( char * message , char * add_mess )
{
 printf(" %s %s \n",message,add_mess );
}

main()
{
 int      error_count, warning_count ;
 FILE   * outfile ;
 FILE   * in_file ;
 char     fn[MAXPATH],outname[MAXPATH], temp[MAXPATH], *ptr;
 char drive[MAXDRIVE] , dir[MAXDIR], file[MAXFILE], ext[MAXEXT];
 error_count = 0 ;
 warning_count = 0 ;
 e_printf = assembler_print_errors ;
 puts(" Generic Assembler. Version 1.0\n");
 puts(" Written by : William E. Ives");
 puts(" Copyright (c) 1988 by Michigan Technological University.\n");
 printf(" Absolute or Relative assembly ? (A/R) ");
 fn[0] = getche();
 putchar('\n');
 am_assem_class = (( fn[0] == 'a')||(fn[0] == 'A')) ?
                  am_absolute : am_relative ;
 printf(" Enter source file name [.ASM] =>");
 fn[0]=MAXPATH-1;
 ptr=cgets(fn);
 strcpy(fn,ptr);
 putchar('\n');
 fnsplit( fn,drive,dir,file,ext);
 if ( ! ( *ext ) )
    fnmerge( fn, drive,dir,file,".ASM");
 fnmerge( outname,drive,dir,file,".LST");
 printf(" Enter name of list file [%s] =>",outname);
 temp[0]=MAXPATH-1;
 ptr = cgets(temp);
 putchar('\n');
 if ( temp[1] ) strcpy(outname,ptr);
 in_file = fopen( fn,"r");
 if ( in_file == NULL ) {
    e_message(0,30, fn );
    return 30 ;
    }
 puts(" Assembling..");
 e_hold_messages = TRUE ;
 am_pass1( in_file , &error_count, &warning_count );
 e_hold_messages = FALSE ;
 fclose(in_file);
 printf(" Total Errors %d    Total Warnings %d \n",
         error_count, warning_count );
 puts(" Writing listing file.\n");
 outfile = fopen( outname,"w");
 if ( outfile == NULL ) {
    e_message(0,30, outname );
    return 30 ;
    }
 l_printlisting( outfile ,TRUE );
 fprintf(outfile,"\n Total Errors %d    Total Warnings %d \n",
                 error_count, warning_count );
 fprintf(outfile,"\n\n Name of file :%s\n",fn);
 fclose(outfile);
 return 0 ;
} /* main */
```

End Listing One

Listing Two

```
/*  68ASSEM.C 68000 Assembly Module. Contains procedures to handle assembly.*/

  #include <stdio.h>
  #include <stdlib.h>
  #include <string.h>
  #include <ctype.h>

  #include "68defs.h"
  #include "68err.h"
  #include "68parse.h"
  #include "68list.h"
  #include "68assem.h"
  #include "68symtab.h"
  #include "68pseudo.h"
  #include "68instr.h"
  #define LINELEN 80
  #define LOCAL near pascal

  void p_assem_line ( char          * line          ,
                      char            label[MAXSYMLEN]  ,
                      char            command[MAXSYMLEN],
                      p_size_type   * size          ,
                      char          * numterms      ,
                      am_term_type ** termlist      ) ;

unsigned long int    am_location_counter = 0L    ;
char                 am_end_found        = FALSE ;
char                 am_trunc_lines      = TRUE  ;
unsigned long int    am_relbase          = 0L    ;
char                 am_relknown         = FALSE ;
                        /* size of absolute address in words. */
                        /* 1 - abs short, 2 - abs long.       */
char                 am_abs_address_size  = 1;
am_term_type       * am_term_list_head = NULL , * am_term_list_tail = NULL ;
am_assem_type        am_assem_class    = am_absolute ;
int am_resolve_symbol( p_sym_type * symlist )
{
  register int   symcount = 0 ;
  unsigned long  sum = 0L ;
  char           rel , relcount = 0 ;
  p_sym_type   * symbol , * temp , * prev ;
  temp = NULL ;
  prev = NULL ;
  symbol = symlist ;
  while ( symbol ) {
     rel = ( symbol->sym[0] == '*'  ) ;  /* set relative flag.*/
     if ( !rel && symbol->sym[0]  ) {      /* if there is a symbol.*/
       symcount++ ;
       prev = symbol ;
       symbol = symbol->next ;
       }
     else {                        /* there is a value. perform calculations */
       if ( symbol->operator == '+' ) {
          if ( rel ) relcount += symbol->sym[1] ;
          sum += symbol->val ;
          }
       else {
          if ( rel ) relcount -= symbol->sym[1] ;
          sum -= symbol->val ;
          symbol->operator = '+';
          }
       if ( !temp ){              /* if temp == null */
          temp = symbol ;
          prev = symbol ;
          symbol = symbol->next ;
          }
       else {
          prev->next = symbol->next ;
          free(symbol);
          symbol = prev->next ;
          }
       }
     }
  if ( !symcount && relcount )
     if ( am_relknown ) {
        sum += ( relcount * am_relbase ) ;   /* resolve relative if known.*/
        relcount = 0;
        }
     else
        symcount=1 ;
  if ( temp ) {
     temp->val = sum ;
     temp->operator = '+' ;
     if ( !relcount ) temp->sym[0] = 0 ;
     else {
        temp->sym[0] = '*' ;
        temp->sym[1] = relcount ;
        }
     }
  if ( prev )  prev->next = NULL ;
  return symcount ;
} /* am_resolve_symbol */
int am_resolve_term ( am_term_type * termlist , char   datatermcount )
{
  register int    termcount = 0 ;
  am_term_type  * term ;
  char            okay , count ;
  term = termlist ;
  okay = TRUE ;
  count = 1 ;
  while ( term && okay ) {
     if ( am_resolve_symbol(term->symptr ) ) termcount++ ;
     term = term->next ;
     count++ ;
     if ( term )
        okay = ( termlist->class == am_first_instr_term &&
                 term->class == am_other_instr_term ) ||
               ( termlist->class == am_data_term &&
                 term->class == am_data_term  &&
                 count       <= datatermcount  ) ;
     }
  return termcount ;
} /* am_resolve_term */
void am_delete_terms ( am_term_type ** termlist , char   all )
{
 am_term_type * term ;
 p_sym_type   * sym  ;
 term = *termlist ;
 if ( all ) {
    while ( term = *termlist ) {
      if (  ((term->modereg >> 8) == 7) &&
            ((term->modereg & 15)==10 ) ) {
         free(term->symptr);                  /* free the string. */
         term->symptr = NULL ;
         }
      while ( sym = term->symptr ) {
         term->symptr = term->symptr->next ;
         free(sym) ;
         }
      *termlist = term->next ;
      free(term);
      }
   }
 else {
   if ( term ) {
     if (  ((term->modereg >> 8) == 7) &&
```

(continued on page 94)

Listing Two *(Listing continued, text begins on page 50.)*

```
          ((term->modereg & 15)==10 ) ) {
        free(term->symptr);                /* free the string. */
        term->symptr = NULL ;
        }
     while ( sym = term->symptr ) {
        term->symptr = term->symptr->next ;
        free(sym) ;
        }
     *termlist = term->next ;
     free(term);
     }
   }
} /* am_delete_terms */
void am_add_terms_to_list ( am_term_type ** termlist  )
{
 am_term_type * term ;
 if ( ! (*termlist) ) return ;           /* if NULL then leave. */
 if ( !am_term_list_tail )  {            /* list is empty */
    am_term_list_head = *termlist ;
    am_term_list_head->prev = NULL ;
    }
 else {
    am_term_list_tail->next = *termlist ;
    am_term_list_tail->next->prev = am_term_list_tail ;
    }
 for ( term = *termlist ; term ; term = term->next )  /* set tail */
     am_term_list_tail = term ;
 *termlist = NULL;
 return ;          /* return used to avoid compilier warning. */
} /* am_add_terms_to_list */
void am_remove_terms_from_list ( am_term_type ** termlist  )
{
 am_term_type * term ;
 char           i    ;
 if ( ! (*termlist) ) return ;           /* if NULL then leave. */
 term = *termlist ;
 i = 1 ;
 if ( term->class == am_first_instr_term ) {
    if ( term->next )
       if ( term->next->class == am_other_instr_term )
          i = 2 ;
    }
 if ( term == am_term_list_head )
    am_term_list_head = ( i == 1 ) ? term->next : term->next->next ;
 else
    term->prev->next = ( i == 1 ) ? term->next : term->next->next ;
 if ( ( term == am_term_list_tail ) ||
      ( ( i == 2 ) && ( term->next == am_term_list_tail )) )
    am_term_list_tail = term->prev  ;
 else
   if ( i == 1 )
     term->next->prev = term->prev ;
   else
     if ( term->next->next ) term->next->next->prev = term->prev ;
 if ( i == 1 )
    term->next = NULL ;
 else
    term->next->next = NULL ;
 am_delete_terms( &term, TRUE );
 *termlist = NULL;
 return ;          /* return used to avoid compilier warning. */
} /* am_remove_terms_from_list */
 void  am_backfill(  am_term_type  * termlist )
 {
  char            i,j,k, reg    ;
  am_term_type *  term ;
  unsigned int    opcode ;
  l_line_type  *  lptr ;
  lptr = termlist->lineptr ;
  opcode = termlist->opcode;   /* get opcode */
  /* write opcode to listing */
  l_writetoline(1,opcode,0,lptr);
  term = termlist ;
  j =  ( term->index == 0 ) ? 2 : 1 ;
  reg = ( opcode & 7 ) ;
  for (i=1,k=2 ; i <= j && term ; i++ ) {
    switch ( reg ) {
      case 1 :     /* abs.l */
         l_writetoline(k,term->symptr->val>>16,0,lptr);
         k++ ;
      case 0 :     /* abs.w */
         l_writetoline(k,term->symptr->val & 0xFFFF ,0,lptr);
         k++ ;
         break ;
       }
      term = term->next ;
      reg = ((opcode>>9)&7) ;
     }
  } /* am_backfill */
static int LOCAL am_readln ( FILE * infile, int linelen,
                             char * line  , char * line1, char * blank )
{
   int   ch, ch1 , i  ;
   char  dqoute, sqoute ; /* flags for tracking double & single quotes */
   dqoute = sqoute = FALSE ;
   *blank = TRUE ;
   i = 0 ;
   while ( ch=ch1=fgetc(infile) ) {
      if ( ch == '\t' ) {                  /* expand tabs into 8 spaces */
         for ( ch = 0 ; ch < 8 ; ch ++, i++ ) {
            if ( i < linelen ) {
               line[i] = line1[i] = ' ';
               line[i+1] = line1[i+1]= NULL;
               }
            }
         continue ;
         }
      if ( !isprint(ch) ) break ;
      if ( ch == '\'' ) { if ( !dqoute ) sqoute = !sqoute ; }
      else if ( ch == '"' ) { if ( !sqoute ) dqoute = !dqoute ; }
      else if (!( dqoute || sqoute ))  ch1 = toupper(ch) ;
      if ( i< linelen ) {
          line[i]= ch ;
          line1[i]= ch1 ;
          if ( *blank ) *blank = ( ch == ' ' );
          line[i+1] = line1[i+1] = NULL ;
          }
      i++;
      }
   if ( i )   return 0 ;
   return ch ;
} /* am_readln */
void am_pass1 ( FILE * in_file , int  * error_count , int  * warning_count )
{
int             i  ;
  ps_pseudos       pseudo_class ;
  char             label[MAXSYMLEN], command[MAXSYMLEN] ;
  char             sizeinwords , blank ;
  p_size_type      size  ;
  p_sym_type     * sym   ;
  char             numterms ;
  am_term_type   * termlist, *term ;
  unsigned int     opcode ;
  unsigned int     index  ;
  char             prev_warnings = 0 ;
  char             line[LINELEN], linehold[LINELEN] ;
  l_line_type    * lptr  ;
  e_error.state = 0 ;
  e_error.warnings = 0 ;
  while ( !am_end_found && !e_error.out_of_memory &&
          ( am_readln(in_file,LINELEN-1,line,linehold,&blank) != EOF )) {
     if ( blank ){                                        /* skip blank lines */
        l_addline( l_neither, 0, "", &lptr );
        continue ;
        }
     if ( line[0] == '*' || line[0] == ';' ) {   /* skip comment lines */
        l_addline( l_neither, 0, line, &lptr );
        continue ;
        }
     *error_count += e_error.state ;  /* count up total number of errors */
     e_error.state = 0 ;
     label[0]   = 0 ;
     command[0] = 0 ;
     size       = p_unknown ;
     numterms   = 0 ;
     sizeinwords= 0 ;
     termlist   = NULL ;
     p_assem_line ( linehold, label, command, &size, &numterms, &termlist );
     if ( e_error.state ) {
        /* add errors to listing and go on to next line */
        l_addline( l_neither, 0, line, &lptr);
        am_delete_terms(&termlist, TRUE );
        l_add_errors(lptr);
        continue ;        /* skip to next source line. */
        }
     /* if command is psuedo then handle it. */
     if ( ps_lookup_pseudo ( command, &pseudo_class ) ) {
        ps_pseudo( label, pseudo_class, numterms, &termlist, line);
        if ( ( e_error.warnings > prev_warnings ) || e_error.state )
           l_add_errors(l_line_head->prev);
        prev_warnings = e_error.warnings ;
        continue ;   /* skip to next source line. */
        }
     if ( am_location_counter & 1 ) {
        e_message(0,22,NULL);             /* location counter is odd */
        l_addline(l_neither, 0, line, &lptr);
        prev_warnings = e_error.warnings ;
        l_add_errors(lptr);
        am_location_counter++; /* adjust counter so error does not repeat*/
        continue ;
        }
     /* its either an instruction or an error */
     if ( numterms > 2 ) {
        e_message(0,15,NULL) ;                /* too many terms on line.*/
        l_addline(l_neither, 0, line, &lptr);
        prev_warnings = e_error.warnings ;
        l_add_errors(lptr);
        continue ;
        }
     /* set size and initial opcode. */
     i = 1 ;
     switch ( numterms ) {
      case 0 :                    /* make source and dest empty */
           i = is_validate ( &size , command, 7, 5, 7, 5,
                       &opcode , &sizeinwords, termlist , &index );
           break;
      case 1 :                    /* make source empty */
           i = is_validate ( &size , command, 7, 5,
                       termlist->modereg >> 8,
                       termlist->modereg & 15,
                       &opcode , &sizeinwords, termlist , &index );

           break;
      case 2 :
           i = is_validate ( &size , command,
                       termlist->modereg >> 8,
                       termlist->modereg & 15,
                       termlist->next->modereg >> 8,
                       termlist->next->modereg & 15,
                       &opcode , &sizeinwords, termlist , &index );
           break;
      }
     if ( i ) {
       l_addline(l_neither, 0, line, &lptr);
       prev_warnings = e_error.warnings ;
       l_add_errors(lptr);
       continue ;
       }
     /* process line label if there is one */
     if ( label[0] )
```

(continued on page 96)

Listing Two *(Listing continued, text begins on page 50.)*

```
        if (sym_add_label_symbol(label,am_location_counter,am_assem_class)){
           l_addline(l_neither, 0, line, &lptr);
           l_add_errors(lptr);
           prev_warnings = e_error.warnings ;
           am_delete_terms(&termlist, TRUE) ;
           continue ;                  /* skip to next source line. */
           }
      /* resolve already known symbols */
      for ( term = termlist ; term ; term = term->next )
        for ( sym = term->symptr ; sym ; sym = sym->next )
           if ( sym->sym[0] )
              sym_add_operand_symbol( sym, term, TRUE );
      l_addline( l_firstinstr, sizeinwords, line, &lptr  );
      if ( e_error.out_of_memory ) {
         am_delete_terms(&termlist,TRUE);
         break ;
         }
      /* its either an instruction or an error */
      if ( e_error.warnings > prev_warnings )
         l_add_errors(lptr);
      prev_warnings = e_error.warnings ;
      if (termlist) {
         termlist->lineptr = lptr ;          /* hold onto line.*/
         if ( termlist->next )
            termlist->next->lineptr = lptr ;
         termlist->index   = index     ;  /* hold onto index.*/
         termlist->opcode  = opcode    ;  /* hold onto opcode.*/
         }
      l_writetoline(0,am_location_counter,0,lptr);
      l_writetoline(sizeinwords,0,3,lptr);       /* put '?' in listing.*/
      /* are all terms knowm? */
      if ( ! am_resolve_term(termlist,0) ) {
         am_backfill( termlist );
         if ( e_error.warnings > prev_warnings )
            l_add_errors(lptr);
         prev_warnings = e_error.warnings ;
         am_location_counter += ( sizeinwords << 1 ) ;
         am_delete_terms(&termlist, TRUE) ;
         }
      else {
         am_add_terms_to_list( &termlist );
         am_location_counter += ( sizeinwords << 1 ) ;
         }
      }
   if ( e_error.out_of_memory ) {
         l_add_errors(l_line_head->prev);
         *error_count += e_error.state ;
         }
   /* Make sure all local and global symbols are resolved */
   e_error.state = 0 ;
   *error_count +=  sym_process_unresolved_locals() ;
   *warning_count = e_error.warnings ;
   /* Add symbol table to listing */
   sym_add_symtabtolisting();
  } /* am_pass1 */
```

End Listing Two

Listing Three

```
/*  68ERR.C - 68000 error handler module */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "68defs.h"
#include "68err.h"
 struct e_struct e_error = { 0,0,0,0,FALSE, NULL, NULL };
 char e_hold_messages = TRUE ;
 e_printf_type e_printf ;
 /* Global error list array. */
 err_type   err_list[MAXERRORS] =
    {{ 1  , 0,"Error 1. Unexpected end of line." ,NULL,NULL },
     { 2  , 0,"Error 2. Unexpected symbol.",NULL,NULL },
     { 3  , 0,"Error 3. Unexpected token char.",NULL,NULL },
     { 4  , 0,"Error 4. Symbol/Literal contains invalid char.",NULL,NULL },
     { 10 , 0,"Error 10. Command op size invalid.",NULL,NULL },
     { 11 , 0,"Error 11. Invalid address mode.",NULL,NULL },
     { 12 , 0,"Error 12. Unrecognized command.",NULL,NULL },
     { 13 , 0,"Error 13. Command operands required.",NULL,NULL},
     { 14 , 0,"Error 14. Forward references not allowed here.",NULL,NULL},
     { 15 , 0,"Error 15. Too many operands.",NULL,NULL},
     { 16 , 0,"Error 16. Line label required.",NULL,NULL},
     { 17 , 0,"Error 17. Label found in external list.",NULL,NULL},
     { 18 , 0,"Error 18. Label already resolved.",NULL,NULL},
     { 19 , 0,"Error 19. Operand symbol already resolved.",NULL,NULL},
     { 21 , 0,"Error 21. Address collision.",NULL,NULL},
     { 22 , 0,"Error 22. Location counter is odd.",NULL,NULL},
     { 23 , 0,"Error 23. Unresolved symbol.",NULL,NULL},
     { 30 , 0,"Error 30. File not found or unable to open.",NULL,NULL },
     { 31 , 0,"Error 31. Unexpected end of file.",NULL,NULL },
     { 33 , 0,"Error 33. Disk full while writing file.",NULL,NULL},
     { 41 , 0,"Error 41. Out of Dynamic memory.",NULL,NULL },
     { 50 , 0,"Warning 50. Symbol too long. Truncated.",NULL,NULL},
     { 52 , 0,"Warning 52. Value out of range.",NULL,NULL},
     { 70 , 0,"Warning 70. Expression/Syntax imprecise.",NULL,NULL},
     { 71 , 0,"Warning 71. Command op size not specified.",NULL,NULL},
     { 72 , 0,"Warning 72. Line label not allowed. Ignored.",NULL,NULL},
     { 73 , 0,"Warning 73. Command operands not allowed. Ignored.",NULL,NULL},
     { 74 , 0,"Warning 74. Symbol already in global list. Ignored.",NULL,NULL},
     { 75 , 0,"Warning 75. Symbol already in external list. Ignored.",NULL,NULL},
     { 100, 0,"FSE 100. Invalid return from next_token.",NULL,NULL} };
 void  e_message( char   pos     ,
                  char   code    ,
                  char * add_mess)
```

```
{
 err_type   * temp ;
 int          i    ;
 char * tmp;
 temp = NULL ;
 if ( e_hold_messages ) {
    temp = ( err_type * ) malloc ( sizeof(err_type) ) ;
    if ( !temp ) code = 41 ;  /* change code to that of 'out of memory' */
    }
 i = 0 ;
 while (( i < MAXERRORS ) && ( err_list[i].code != code )) i ++ ;
 if ( code == 41 )
    e_error.out_of_memory = TRUE ;
 if ( temp ) {
    temp->message = ( i <= MAXERRORS ) ? err_list[i].message : NULL ;
    if ( *add_mess )
       temp->add_mess = ( char * ) strdup ( add_mess ) ;
    else
       temp->add_mess = NULL ;
    temp->code    = code ;
    temp->position= pos ;
    temp->next = NULL ;
    if ( !e_error.errptr ) {
       e_error.errptr = temp ;
       e_error.last   = temp ;
       }
    else {
      e_error.last->next = temp ;
      e_error.last = temp ;
      }
    }
 tmp = ( char * ) ( i < MAXERRORS ) ? err_list[i].message : NULL ;
 if ( e_error.curpos )
   pos += e_error.curpos ;
 e_printf( tmp, add_mess );
 if ( ( code < 50 ) || ( code >= 100 ) ) {
    e_error.errors++ ;
    e_error.state++ ;
    }
 else
    e_error.warnings++;
} /* e_message */
void e_delete_errors()
{
 err_type  * temp ;
 while ( e_error.errptr ) {
    temp = e_error.errptr->next ;
    if ( e_error.errptr->add_mess )
       free( e_error.errptr->add_mess ) ;
    free( e_error.errptr );
    e_error.errptr = temp ;
    }
} /* e_delete_errors */
```

End Listing Three

Listing Four

```
/* 68INSTR.C - 68000 Instruction Set Module contains procedures to handle
               instruction set.*/

  #include <stdio.h>
  #include <stdlib.h>
  #include <string.h>

  #include "68defs.h"
  #include "68err.h"
  #include "68parse.h"
  #include "68list.h"
  #include "68assem.h"
  #include "68instr.h"

  #define LOCAL near pascal
  int   is_validate ( p_size_type *  size   ,
                      char        *  command,
                      char           smode  ,
                      char           sreg   ,
                      char           dmode  ,
                      char           dreg   ,
                      unsigned int * opcode ,
                      char         * total_size ,
                      am_term_type * termlist ,
                      unsigned int * index    )
  {
    /* error if destination is not specified */
    if ( (dmode==7) && ( dreg==5 ) ) {
        e_message(0,11," Destination operand required.");
        return 11 ;
        }
    *total_size = 1 ;
    /* lookup command */
    if (  ! strcmp(command,"MOVE")  ) {
       /* error if source is not specified */
       if ( (smode==7) && ( sreg==5 ) ) {
          e_message(0,11," Source operand required.");
          return 11 ;
          }
       if ( *size == p_unknown ) {
          /* assign default size if it's unknown */
          e_message(0,71," Assumed Long.");
          *size = p_long ;
          }
       termlist->sizeofreserve=am_abs_address_size ;
       termlist->next->sizeofreserve=am_abs_address_size ;
```

(continued on page 98)

Listing Four *(Listing continued, text begins on page 50.)*

```
        *total_size += ( am_abs_address_size << 1 );
        *opcode =  (dreg << 9 ) ¦ ( dmode <<6 ) ¦ ( smode <<3 ) ¦ sreg ;
        *opcode ¦= ( *size == p_long )? 0x2000: (*size==p_word)? 0x3000:0x1000;
        *index = 0;
        }
     else if (  ! strcmp(command,"JMP")  ) {
        *opcode =  0x4EC0 ¦ ( dmode << 3 ) ¦ dreg ;
        termlist->sizeofreserve=am_abs_address_size ;
        *total_size += am_abs_address_size ;
        *index = 1 ;
        }
     else {
        e_message(0,12,command);
        return 12 ;
        }
     return 0 ;
   } /* is_validate */
```

End Listing Four

Listing Five

```
/* 68LIST.C - 68000 listing module. */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <ctype.h>

#include "68defs.h"
#include "68err.h"
#include "68list.h"
l_line_type *   l_line_head = NULL ;
char        *  l_header    =
" Generic Assembler Version 1.0 :                          ";
char        *  l_blanks    = "                          " ;
int             l_number_of_lines = 0 ;
int  l_addline( l_lclass_type     class      ,
                char              numofwords ,
                char           * line       ,
                l_line_type    ** lineptr    )
{
 l_line_type  * curline , * tcurline ;
 register int   i ;
 long           secs ;
 *lineptr = NULL ;
 curline = NULL ;
 curline = ( l_line_type * ) malloc ( sizeof(l_line_type) ) ;
 if ( !curline ) {
    e_message(0,41," Listing " ) ; /* out of memory */
    return 41 ;
    }
 curline->lclass = class ;
 curline->linenum = ( class != l_firstinstr ) ? 0 :l_number_of_lines++ ;
 if ( !l_line_head ) {
    l_line_head = ( l_line_type * ) malloc ( sizeof(l_line_type));
    if (!l_line_head){
      e_message(0,41 ," Listing " ) ; /* out of memory */
      return 41 ;
      }
    l_line_head->line = l_header ;
    time(&secs);
    strcpy ( ((l_line_head->line)+32), asctime( localtime(&secs) ) );
    i = strlen(l_line_head->line);
    for(;!isprint(l_line_head->line[i]);i--) l_line_head->line[i]=0;
    l_line_head->lclass = l_data ;
    l_line_head->linenum = 0 ;
    l_line_head->next = l_line_head ;
    l_line_head->prev = l_line_head ;
    }
 curline->line = ( char * ) malloc(  25 + strlen(line) );
 if ( !(curline->line) ) {
    e_message(0,41 ," Listing " ) ; /* out of memory */
    free(curline);
    return 41 ;
    }
 curline->next = l_line_head ;
 curline->prev = l_line_head->prev ;
 l_line_head->prev = curline ;
 curline->prev->next = curline;
 if ( class != l_neither ) {
    memset( (curline->line) ,'0', 6 );
    memset( ((curline->line)+6),' ',18 );
    }
 else
    memset( (curline->line) ,' ',24);
 strcpy( ((curline->line)+24) ,line);
 tcurline = curline ;
 if ( numofwords > 3 ) {
    i = (numofwords - 3) / 3 ;
    if ( (numofwords - 3) % 3  ) i++ ;
    for ( ; i ; i-- ) {
       curline = ( l_line_type * ) malloc ( sizeof(l_line_type) ) ;
       if ( !curline ) {
          e_message(0,41 , NULL ) ; /* out of memory */
          return 41 ;
          }
       curline->next = l_line_head ;
       curline->prev = l_line_head->prev ;
       l_line_head->prev = curline ;
       curline->prev->next = curline;
       curline->line = strdup( l_blanks );
       if ( !(curline->line) ) {
          e_message(0,41 , NULL ) ; /* out of memory */
          return 41 ;
          }
       curline->lclass = ( class == l_firstinstr ) ? l_otherinstr:l_data ;
       curline->linenum =( class == l_firstinstr ) ? l_number_of_lines++ : 0 ;
       }
    }
 *lineptr = tcurline ;
 return 0 ;
} /* l_addline */
void    l_writetoline( int              spec       ,
                       unsigned long    value      ,
                       char             option     ,
                       l_line_type    * line       )
{
 register int  i       ;
 char        * curline ;
 if ( !line ) return ;
 if ( !spec ) {
    if ( option )
       strnset(line->line,'?',6);
    else {
       sprintf(line->line,"%06X",value);
       *(line->line+6) = ' ' ;
       }
    }
 else {
    i = ( spec - 1 ) / 3 ;
    for (  ; i ; i-- , line = line->next ) {
        if ( option == 2 ) {
           sprintf(line->line+6,"   %04X %04X %04X",( int ) value,
                   ( int ) value, ( int ) value );
           *(line->line+23) = ' ';
           }
        else if ( option == 3 )  {
           sprintf(line->line+6,"   ???? ???? ????" );
           *(line->line+23) = ' ';
           }
        }
    i = ( spec - 1 ) % 3  ;
    curline = line->line + 9 ;
    for ( ; i ; i-- , curline += 5 ) {
        if ( option == 2 )  {
           sprintf( curline,"%04X",( unsigned int ) value);
           *(curline+4) = ' ';
           }
        else if ( option == 3 )
           strnset( curline,'?',4 );
        }
    if ( ( option == 0 ) ¦¦ ( option == 2 ) )  {
       sprintf(curline,"%04X", ( unsigned int ) value ) ;
       *(curline+4)=' ';
       }
    else if ( option == 4 ) {
       sprintf(curline,"%02X", ( unsigned char ) value );
       *(curline+2)=' ';
       }
    else
       strnset(curline,'?',4);
    }
} /* l_writetoline */
void    l_add_errors ( l_line_type * lptr )
{
  l_line_type   * tlptr ;
  err_type      * error ;
  char            relink_list ;
  char            message_buf[100] ;
  if ( !lptr )  /* if lptr == NULL */
     lptr = l_line_head->prev ;
  /* if lptr at end of list do not bother relinking listing.*/
  relink_list = ( lptr->next != l_line_head ) ;
  for ( error = e_error.errptr ; error ; error = error->next ) {
     strcpy( message_buf , error->message ) ;
     strcpy( message_buf+strlen(error->message), error->add_mess );
     l_addline(l_neither, 0, message_buf , &tlptr );
     if ( relink_list ) {
        /* unlink the  line */
        tlptr->next->prev = tlptr->prev ;
        tlptr->prev->next = tlptr->next ;
        /* link the line into listing after the source line at lptr */
        tlptr->next = lptr->next ;
        tlptr->next->prev = tlptr ;
        tlptr->prev = lptr ;
        lptr->next = tlptr ;
        }
     }
  e_delete_errors() ;
 } /* l_add_errors */
int  l_printlisting( FILE * outfile, char withheader )
{
 l_line_type * curline ;
 if ( withheader )
    fprintf(outfile,"%s\n",l_line_head->line);
 for ( curline = l_line_head->next ; curline != l_line_head ;
       curline = curline->next ) {
      if ( (curline->lclass != l_neither)&&(curline->lclass != l_data) ) {
        if ( fprintf(outfile,"%3d %s\n",curline->linenum,
                     curline->line) == EOF ) {
           e_message(0,33,NULL);
           return 33 ;
           }
        }
      else if ( fprintf(outfile,"    %s\n", curline->line) == EOF ) {
         e_message(0,33,NULL);
         return 33 ;
         }
      }
 return 0 ;
} /* l_printlisting */
void l_delete_listing( char deletehead , char resetcount )
{
 l_line_type * curline, * tline ;
```

(continued on page 100)

Listing Five *(Listing continued, text begins on page 50.)*

```
if ( resetcount ) l_number_of_lines = 0 ;
if ( l_line_head ) {
   l_line_head->prev->next = NULL ;
   curline = l_line_head->next ;
   l_line_head->next = l_line_head ;
   l_line_head->prev = l_line_head ;
   if ( deletehead ) {
      free(l_line_head);
      l_line_head = NULL ;
      }
   while ( curline ) {
         tline = curline->next ;
         if ( curline->line ) free( curline->line );
         free( curline );
         curline=tline ;
         }
   }
} /* l_delete_listing */
```

End Listing Five

Listing Six

```
/* 68MATH.C - 68000 math module. */

#include <stdio.h>
#include <string.h>
#include <ctype.h>

#include "68defs.h"
#include "68err.h"
#include "68parse.h"

unsigned long int strtolong ( register char     *  str    ,
                              char              ** endptr ,
                              char                 base   )
{
   unsigned long int  sum  = 0L ;
   register char shift ;
   switch ( base ) {
     case 10 : for (; *str ; str++)
                 if ( isdigit(*str) )
                    sum = ( sum * 10 ) + ( *str - '0' ) ;
                 else {
                    *endptr = str ;
                    return 0 ;
                    }
               *endptr = str ;
               return sum ;
     case 2  : shift = 1 ; break ;
     case 8  : shift = 3 ; break ;
     case 16 : shift = 4 ; break ;
     }
   for (; *str ; str++ )
      if ( isdigit ( *str ) &&
           (  ( base == 10 ) || ( base == 16 ) ||
              ( (base == 2) && ( *str == '0' || *str == '1' ) ) ||
              ( (base == 8) && ( *str <= '7' && *str >= '0' ) ) ) )
         sum  = (  sum << shift  ) | ( *str - '0' ) ;
      else if ( isxdigit (*str) && ( base == 16 ) )
         sum  = ( sum << shift ) | ( toupper(*str) - 'A' + 10 ) ;
      else {
         *endptr = str ;
         return 0 ;
         }
   *endptr = str ;
   return sum ;
} /* strtolong */
int      m_symtoval( char * sym , unsigned long int  * value )
{
  char    symbol[MAXSYMLEN] ;
  char    ch , * last , base ;
  strncpy( symbol, sym, MAXSYMLEN );  /* make a local pass by value symbol*/
  symbol[MAXSYMLEN-1] = 0 ;
  last = symbol+strlen(symbol)-1 ;
  *value = 0L ;
  if ( isdigit( ch=*symbol ) )
     switch ( toupper( *last )  ) {
       case 'H' : *last = 0 ; base = 16 ; break ;
       case 'B' : *last = 0 ; base = 2  ; break ;
       case 'Q' :
       case 'O' : *last = 0 ; base = 8  ; break ;
       case 'D' : *last = 0 ;
       default  :  base = 10 ; break ;
       }
  else {
     switch ( ch ) {
        case '$' : symbol[0] = '0' ; base = 16 ; break ;
        case '@' : symbol[0] = '0' ; base = 8 ; break ;
        case '%' : symbol[0] = '0' ; base = 2 ; break ;
        case '\'':
                   /* scan line until next ' */
                   last = strchr( symbol+1, '\'');
                   if ( !(*last) ) {
                      e_message(0,51,"End quote expected.");
                      last = symbol+5 ;
                      }
                   *last = 0 ;
                   if ( last - symbol  > 5 ) {
                       e_message(0,51,NULL);
                       symbol[5] = 0 ;
                       }
                   for ( last = symbol+1 ; *last ; last++ )
                       *value = ( *value << 8 ) | *last ;
                   return 0 ;
        default  : return 1 ;
        }
     }
     *value = ( unsigned long int ) strtolong( symbol, &last , base );
     if ( *last ) e_message(0,4,NULL);
     return  ( *last  ) ? 4 : 0 ;
} /* m_symtoval */
```

End Listing Six

Listing Seven

```
/* 68 PARSE.C - 68000 parser module. */

#include <stdio.h>
#include <alloc.h>
#include <string.h>
#include <ctype.h>

#include "68defs.h"
#include "68err.h"
#include "68parse.h"
#include "68list.h"
#include "68assem.h"
#include "68math.h"

#define LOCAL near pascal
   typedef enum { symbol, token, none } p_tok_type ;
   typedef enum {  pn_sym   ,        /* sym */
                   pn_empty ,        /* empty parse node. */
                   pn_error          /* error */
                } p_addr_type ;
   typedef struct {
                   p_addr_type   p_nodeclass;/* tag field */
                   char          regnum   ;
                   p_sym_type *  symptr  ; /* symbol list, if any. */
                  } p_node_type ;
    char          symchars[38] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890_" ;
    p_sym_type    curtoksym ;      /* current token symbol from next_token */
    char          curtok ;         /* current token character from next_token*/
 static p_tok_type LOCAL next_token( char * line     , int  * line_offset  ,
                                     int    line_len );
 static int LOCAL  p_symbol ( char *      line     , int  * line_offset   ,
                              int         line_len , char   symknown      ,
                              char        preop    , p_sym_type ** symptr ,
                              p_node_type * p_node   );
 static p_size_type LOCAL p_dotsize ( char * line    ,
                                      int  * line_offset ,
                                      int    line_len ,
                                      int    prev_offset );
 static void LOCAL p_parse_line ( char *   line     , int  * line_offset    ,
                                  int      line_len , p_node_type *  p_node ,
                                  char *   done     );
 static p_tok_type LOCAL next_token( char * line     , int  * line_offset  ,
                                     int    line_len )
{
  char         * tokpos, * tline ;
  int          symlen  ;
  (*line_offset) += strspn( line + *line_offset , " \t");  /* skip blanks */
  if ( *line_offset >= line_len )
     return none ;
  tline = line + *line_offset ;
  if ( *tline == '\'') {
     tokpos = strchr( tline+1, '\'');      /* scan for next quote.*/
     if ( *tokpos )                        /* if found set tok one further. */
        tokpos++ ;                         /* if not , set to shortest of   */
     else                                  /* (5 forward or the end)        */
        tokpos = ( tokpos - tline > 5 ) ? tline + 5 : tokpos ;
     }
  else
    tokpos = strpbrk( tline, ".;-+()#/, \"\t");
  if ( tokpos != tline ) {
     /* return symbol between start of line and tokpos. */
     if ( !(*tokpos) ) tokpos = line + line_len ;  /* correct for NULL */
     symlen = tokpos - tline ;
     (*line_offset) += symlen;
     if ( symlen >= MAXSYMLEN ) e_message(*line_offset,50,NULL);
     symlen = ( symlen >= MAXSYMLEN ) ? MAXSYMLEN - 1 : symlen ;
     strncpy( curtoksym.sym, tline, symlen ) ;
     curtoksym.sym[symlen] = 0 ;
     if ( isalpha(curtoksym.sym[0] ) )
        if ( strspn( curtoksym.sym, symchars ) == symlen )
           return symbol ;
        else {
           e_message(*line_offset,4,NULL);
           return symbol ;
           }
     e_error.curpos = *line_offset ;
     if ( ( !m_symtoval(curtoksym.sym,&curtoksym.val ) ) &&
          ( !e_error.state ) )
         curtoksym.sym[0] = 0 ;
     e_error.curpos = 0 ;
     return  symbol ;
     }
  else
     {
     /*  the token is at the start of the line. */
     curtok = *tline ;
     (* line_offset ) ++ ;
     return token ;
     }
} /* next_token */
 static int LOCAL  p_symbol ( char *      line     , int  * line_offset   ,
                              int         line_len , char   symknown      ,
                              char        preop    , p_sym_type ** symptr ,
                              p_node_type * p_node   )
{
  p_tok_type   tok ;
  int          i ;
  if ( symknown )
     tok = symbol ;
  else
```

```
    tok = next_token( line , line_offset , line_len) ;
  switch ( tok ) {
    case symbol : /* Symbol was found */
                  *symptr = ( p_sym_type * ) malloc ( sizeof(p_sym_type));
                  if ( ! (*symptr) ) {   /* if *symptr == NULL */
                     e_message(0,41," Parser ");
                     return 41 ;
                     }
                  memcpy( (*symptr) , &curtoksym, sizeof(p_sym_type) );
                  (*symptr)->operator = preop ;
                  (*symptr)->next = NULL ;
                  i = *line_offset ; /* hold onto line offset */
                  tok = next_token( line , line_offset , line_len);
                  switch ( tok ) {
                    case token :
                         switch ( curtok ) {
                            case '+' :
                            case '-' :
                                i = p_symbol( line, line_offset,
                                              line_len, FALSE, curtok,
                                              &( (*symptr)->next ), p_node);
                                return i ;
                            default  :
                                /* un-get the token. */
                                *line_offset = i ;
                                return 0;
                                }
                    case symbol : e_message(*line_offset,2,NULL);
                                  return 2 ;
                    case none   : return 0 ;
                    default     : e_message(*line_offset,100,"p_symbol");
                                  return 100 ;
                    }
    case token  : e_message(*line_offset,3,NULL);
                  return 3 ;
    case none   : e_message(*line_offset,1,NULL);
                  return 1 ;
    default     : e_message(*line_offset,100,"p_symbol");
                  return 100 ;
    }
} /* p_symbol */
static p_size_type LOCAL p_dotsize ( char * line    ,
                                     int  * line_offset ,
                                     int    line_len ,
                                     int    prev_offset )
{
  p_tok_type  tok     ;
  if ( curtok == '.' ) {
    prev_offset = *line_offset ;
    tok = next_token( line , line_offset, line_len);
    if ( (tok == symbol )&&
         ( curtoksym.sym[1] == NULL ) )
       switch ( curtoksym.sym[0] ) {
          case 'B': return p_byte ;
          case 'W': return p_word ;
          case 'L': return p_long ;
          }
    e_message(prev_offset,70," '.' ignored.");
    *line_offset = prev_offset ;
    return p_unknown ;
    }
  else {
    *line_offset = prev_offset ;
    return p_unknown ;
    }
} /* p_dotsize */
static void LOCAL p_parse_line ( char *   line     , int  * line_offset    ,
                                 int      line_len , p_node_type *  p_node ,
                                 char *   done     )
{
  p_tok_type   tok ;
  tok = next_token( line , line_offset , line_len) ;
  p_node->p_nodeclass = pn_empty ;
  p_node->symptr = NULL ;
  switch ( tok ) {
     case symbol :
                    p_node->p_nodeclass = pn_sym ;
                    p_symbol(line, line_offset, line_len , TRUE,
                             '+',&(p_node->symptr),p_node );
                    break;
     case token  : switch ( curtok ) {
                      case ';' : *done = TRUE ;  /* Comment found */
                                 break;
                      default  : e_message(*line_offset,3,NULL);
                                 *done = TRUE ;
                                 break;
                      }
                    break ;
     case none   : *done = TRUE ;   /* End of line found */
                    break ;
     default     : e_message(*line_offset,100,NULL);
     }
  *done = ( e_error.state ) ? TRUE:*done ;
  if ( ! *done ) {
  tok = next_token( line , line_offset , line_len) ;
  switch ( tok ) {
     case symbol :
           e_message(*line_offset,2,curtoksym.sym); /* Unexpected symbol */
           *done = TRUE ;
           break ;
     case token  :
           switch ( curtok ) {
              case ',' : break;
              case ';' : *done = TRUE ;              /* Comment found */
                         break;
              default  : e_message(*line_offset,3," ',' or ; exp.");
                         *done = TRUE ;
                         break;
```

(continued on page 102)

Listing Seven *(Listing continued, text begins on page 50.)*

```
            }
          break ;
    case none   : *done = TRUE ;   /* End of line found */
                  break ;
    default     : e_message(*line_offset,100,NULL);
                  *done = TRUE ;
    }
  }
  if ( e_error.state ) p_node->p_nodeclass = pn_error ;
} /* p_parse_line */
void p_assem_line ( char           * line          ,
                    char             label[MAXSYMLEN]  ,
                    char             command[MAXSYMLEN],
                    p_size_type    * size          ,
                    char           * numterms      ,
                    am_term_type ** termlist       )
{
  register int   i      ;
  int            line_offset = 0, line_len;
  char           done   , mode,  reg ;
  am_term_type * term , * prev ;
  p_tok_type     tok    ;
  p_node_type    p_node ;            /* current parse node. */
  e_error.state = 0 ;
  /* clear the parameters */
  label[0] = 0 ;
  command[0] = 0 ;
  *size = p_unknown ;
  *numterms = 0 ;
  *termlist = NULL ;
  line_len = strlen(line) ;
  /* read the line label */
  if ( isalpha( line[0] ) ) {
     /* scan the line to remove the label string. */
     tok = next_token( line , &line_offset , line_len) ;
     if (tok == symbol)
        strncpy( label,curtoksym.sym,MAXSYMLEN );
     }
  /* read the command */
  i = line_offset ;
  tok = next_token( line , &line_offset , line_len) ;
  switch ( tok ) {
     case symbol : strncpy(command,curtoksym.sym,MAXSYMLEN);
                   /* check for size identifier */
                   i = line_offset ;
                   tok = next_token( line , &line_offset , line_len) ;
                   switch ( tok ) {
                      case symbol : line_offset = i ; break ;
                      case token  : *size = p_dotsize(line,&line_offset,
                                                      line_len,i);
                                    break;
                      case none   : line_offset = i ; break ;
                      }
                   break ;
     case token  :
     case none   : line_offset = i; break ;
     }
  i = 0 ;
  done = FALSE ;
  while ( ! done ) {
     p_parse_line( line, &line_offset,line_len,&p_node,&done);
     if ( e_error.state ) break ;
     if ( p_node.p_nodeclass == pn_empty ) break ;
     switch ( p_node.p_nodeclass ) {
        case  pn_sym   : mode = 7 ;
                reg = ( am_abs_address_size == 1 )?0:1 ; break ; /* sym  */
        case  pn_empty : mode = 7 ; reg = 5      ; break ; /* empty/none*/
        case  pn_error : mode = 7 ; reg = 11     ; break ; /* error  */
        }
     term = ( am_term_type * ) malloc ( sizeof( am_term_type ) ) ;
     if ( !term ) {
         e_message(0,41,"Parser");   /* not enough memory */
         break ;
         }
     term->modereg = ( mode << 8 ) | reg ;
     term->symptr = p_node.symptr ;
     term->next = NULL ;
     term->prev = NULL ;
     term->sizeofreserve = 0 ;
     term->postword = 0 ;
     term->class = am_other_instr_term ;
     (*numterms)++ ;
     if ( !(*termlist) )  /* if termlist = NULL */
       *termlist = term  ;
     else {
       term->prev = prev ;
       prev->next = term ;
       }
     prev = term ;
     } /* while */
  if ( *termlist )
     (*termlist)->class = am_first_instr_term ;
} /* p_assem_line */
```

End Listing Seven

Listing Eight

```
/* 68PSEUDO.C -  68000 Pseudo Module.                              */
/*               Contains procedures for psuedo command assembly. */

  #include <stdio.h>
  #include <stdlib.h>
  #include <string.h>

  #include "68defs.h"
  #include "68err.h"
  #include "68parse.h"
  #include "68list.h"
  #include "68assem.h"
  #include "68symtab.h"
  #include "68pseudo.h"

  typedef struct  {  char          pseudo[10];
                     ps_pseudos    index  ;
                  } ps_pseudo_type ;
  ps_pseudo_type  ps_pseudo_array [7] =
                           { { "ABS_LONG"  , ps_abslong } ,
                             { "ABS_SHORT" , ps_absshort} ,
                             { "END"       , ps_end     } ,
                             { "EQU"       , ps_equate  } ,
                             { "EXTERN"    , ps_extern  } ,
                             { "GLB"       , ps_global  } ,
                             { "ORG"       , ps_origin  } } ;
  int ps_lookup_pseudo ( char       * pseudo       ,
                         ps_pseudos * pseudo_class )
  {
   ps_pseudo_type * indx ;
   if ( indx = ( ps_pseudo_type * ) bsearch( pseudo, ps_pseudo_array, 7,
                                  sizeof(ps_pseudo_type),strcmp) ){
      *pseudo_class = indx->index ;
      return 1 ;
      }
   else
      return 0 ;
  } /* ps_lookup_pseudo */
  int ps_one_symbol_only ( p_sym_type * symlist )
  {
    if ( symlist )
       if ( symlist->next )
          return FALSE ;
       else
          if ( symlist->sym[0] )
             return TRUE ;
          else
             return FALSE ;
    else
       return FALSE ;
  } /* ps_one_symbol_only */
  int  ps_validate_pseudo ( char          * label    ,
                            char            numterms ,
                            am_term_type ** termlist ,
                            char            onetermonly    ,
                            char            zeroterms      ,
                            char            forwardsallowed,
                            char            labelrequired  ,
                            char            ignorlabel     ,
                            char            stringsallowed ,
                            char            onesymbolonly  )
  {
   am_term_type * term ;
   p_sym_type   * sym  ;
   int            i    , mode, reg  ;
   if ( ( onetermonly && ( numterms > 1 )) ||
        ( zeroterms   && ( numterms     )) ) {
      e_message(0,15,NULL) ;              /* requires one operand */
      i = 15 ;
      goto deleteterms ;
      }
   if ( onetermonly && ( numterms == 0 ) ) {
      e_message(0,13,NULL) ;        /* requires at least one operand */
      return 13 ;
      }
   if ( labelrequired && !(*label) ) {
      e_message(0,16,NULL) ;                /* label required */
      i = 16 ;
      goto deleteterms ;
      }
   if ( ignorlabel && *label )
      e_message(0,72,NULL ) ;                /* label ignored. */
   if ( numterms )
      /* validate each term */
      for  ( term = *termlist ; term ; term= term->next ) {
         mode = term->modereg >> 8 ;
         reg  = term->modereg &  15 ;
         if (  ( mode == 7 ) && ( reg <= 1 ) ) {
            if ( onesymbolonly ) {
               if ( ! ps_one_symbol_only(term->symptr)  ) {
                  e_message(0,11,NULL)  ; /* illegal term */
                  i = 11 ;
                  goto deleteterms ;
                  }
               sym_add_operand_symbol( term->symptr, term, FALSE );
               if ( ( term->symptr->sym[0] == '*' ) ||
                    ( !term->symptr->sym[0] )) {
                  e_message(0,19,NULL);
                  i = 19 ;
                  goto deleteterms ;
                  }
               }
            else {
               for ( sym = term->symptr ; sym ; sym = sym->next ) {
                  if ( sym->sym[0] )
                     sym_add_operand_symbol( sym, term, FALSE );
                  }
               /* compress symbol chain and hold onto number of unresolved */
               /* symbols in postword                                      */
               term->postword = am_resolve_symbol(term->symptr) ;
               if ( ( ! forwardsallowed ) && ( term->postword ) ) {
                  e_message(0,14,NULL) ;         /* cannot forward reference */
                  i = 14 ;
                  goto deleteterms ;
                  }
               }
            }
         else {
            if ( ! ( stringsallowed && ( mode == 7 ) && ( reg == 10 ) )) {
               e_message(0,11,NULL)  ; /* illegal term */
```

```
            i = 11 ;
            goto deleteterms ;
            }
          }
        }
 return 0 ;
 deleteterms :        /* label for exit with deletion of terms */
                      /* i will contain the error code returned*/
 if ( numterms )                    /* delete all terms */
   am_delete_terms( termlist, TRUE );
  return i ;
 } /* ps_validate_pseudo */
 int  ps_pseudo ( char          * label    ,
                  ps_pseudos      index    ,
                  char            numterms ,
                  am_term_type ** termlist ,
                  char          * line     )
 {
  am_term_type * term  ;
  int            i     ;
  l_line_type  * lptr  ;
  switch ( index ) {
     case ps_abslong : case ps_absshort :
     case ps_even    : case ps_end      :
          l_addline( l_neither, 0, line, &lptr); /* add line to listing */
          if ( *label ) e_message(0,72,NULL)    ; /* label ignored. */
          if ( numterms ) e_message(0,73,NULL)  ; /* operands ignored. */
          switch ( index ) {
             case ps_abslong  : am_abs_address_size = 2 ; break ;
             case ps_absshort : am_abs_address_size = 1 ; break ;
             case ps_even     : if ( am_location_counter & 1 )
                                    am_location_counter++ ;
                                break;
             case ps_end      : am_end_found = TRUE ; break ;
             }
          am_delete_terms( termlist, TRUE ) ;
          return 0 ;
     case ps_equate :
          l_addline(l_neither,0,line,&lptr) ;
          /* requires one term only. label required. */
          if ( i = ps_validate_pseudo ( label, numterms, termlist,
                              TRUE,FALSE,FALSE,TRUE,FALSE,FALSE,FALSE))
             return i ;
          /* add label and value to symbol table */
          i = sym_add_label_symbol( label, (*termlist)->symptr->val,
                                    am_absolute ) ;
          am_delete_terms( termlist, TRUE ) ;
          return i ;
     case ps_extern :
          l_addline(l_neither,0,line,&lptr) ;
          /* one or more terms. label ignored. one symbol per each term */
          if ( i = ps_validate_pseudo ( label, numterms, termlist,
                              FALSE,FALSE,FALSE,FALSE,TRUE,FALSE,TRUE))
             return i ;
          /* Add each symbol to the external symbol list. */
          for ( term = *termlist ; term ; term = term->next )
              if ( i = sym_add_extern(term->symptr->sym) ) {
                 am_delete_terms( termlist, TRUE );
                 return i ;
                 }
          am_delete_terms( termlist, TRUE );
          return 0 ;
     case ps_global :
          l_addline(l_neither,0,line,&lptr) ;
          /* one or more terms. label ignored. one symbol per each term */
          if ( i = ps_validate_pseudo ( label, numterms, termlist,
                               FALSE,FALSE,FALSE,FALSE,TRUE,FALSE,TRUE))
             return i ;
          /* Add each symbol to the global symbol list. */
          for ( term = *termlist ; term ; term = term->next )
              if ( i = sym_add_global(term->symptr->sym) ) {
                 am_delete_terms( termlist, TRUE );
                 return i ;
                 }
          am_delete_terms( termlist, TRUE );
          return 0 ;
     case ps_origin :
          l_addline(l_neither,0,line,&lptr) ;
          /* requires one term only. label ignored. */
          if ( i = ps_validate_pseudo ( label, numterms, termlist,
                              TRUE,FALSE,FALSE,FALSE,TRUE,FALSE,FALSE))
             return i ;
          am_location_counter = (*termlist)->symptr->val ;
          am_delete_terms( termlist, TRUE );
          return 0 ;
     }
  return 0 ;
 } /* ps_pseudo */
```

End Listing Eight

Listing Nine

```
/* 68SYMTAB.C - 68000 Symbol table Module. Contains those  */
/*              procedures for symbol table.               */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "68defs.h"
#include "68err.h"
#include "68parse.h"
#include "68list.h"
#include "68assem.h"
#include "68symtab.h"
```

(continued on page 104)

Listing Nine *(Listing continued, text begins on page 50.)*

```
/* Module level variable */
sym_label_type    *  sym_glb_lab_head = NULL ;
sym_label_type    *  sym_local_lab_head = NULL ;
sym_operand_type  *  sym_ext_ref_head = NULL ;
sym_operand_type  *  sym_local_ref_head = NULL ;

/* Local prototypes */
sym_operand_type * sym_lookup_extern( char  *  symbol );
sym_label_type   * sym_lookup_local ( char  *  symbol );
int                sym_addtolocaloplist ( p_sym_type    * symptr  ,
                                          am_term_type  * termptr ) ;
void sym_add_symtabtolisting()
{
  register  sym_label_type   * labptr ;
  sym_operand_type * refptr ;
  l_line_type      * lptr  ;
  char             * message_buf , * chptr , count ;
  int                i ;
  if ( sym_glb_lab_head || sym_local_lab_head || sym_ext_ref_head ) {
     l_addline(l_neither, 0,"",&lptr);
     l_addline(l_neither, 0,
               " Symbol                Value     Class ",&lptr);
     l_addline(l_neither, 0,
               " ----------------      ------    -------",&lptr);
     }
  else
     return ;
  message_buf = "                                                              " ;
  for (count = 1 ; count <= 2 ; count++ ) {
     if ( count == 1 ) {
        strncpy(message_buf+33,"Global ",7);
        labptr = sym_glb_lab_head ;
        }
     else {
        strncpy(message_buf+33,"Local  ",7);
        labptr = sym_local_lab_head ;
        }
     for ( ; labptr ; labptr= labptr->next ) {
        i = strlen( labptr->symbol );
        strncpy( message_buf + 1, labptr->symbol, i );
        for ( chptr = message_buf + i + 1 ; i < MAXSYMLEN ; i++, chptr++ )
            *chptr = ' ' ;
        if ( labptr->relative == '?' ) {
           strncpy( message_buf + 23 ,"??????",6 );
           strncpy( message_buf + 43," Unknown  ", 10 );
           }
        else {
           sprintf( message_buf+ 23 ,"%06lX",labptr->val );
           message_buf[29] = ' ' ;
           if ( labptr->relative == '*' )
              strncpy( message_buf + 43," Relative ", 10 );
           else
              strncpy( message_buf + 43 ," Absolute ", 10 );
           }
        l_addline(l_neither, 0, message_buf , &lptr );
        }
     }
  strncpy( message_buf + 23, "??????",6 );
  strncpy( message_buf + 33, "Extern ",7);
  strncpy( message_buf + 43, " Unknown  ", 10 );
  for ( refptr = sym_ext_ref_head ; refptr ; refptr= refptr->next ) {
      i = strlen( refptr->symbol );
      strncpy( message_buf + 1, refptr->symbol, i );
      for ( chptr = message_buf + i + 1 ; i < MAXSYMLEN ; i++, chptr++ )
          *chptr = ' ' ;
      l_addline(l_neither, 0, message_buf , &lptr );
      }
} /* sym_add_symtabtolisting */
int  sym_process_unresolved_locals()
{
  int                i  ;
  sym_ref_type     * reflistptr ;
  sym_operand_type * refptr ;
  sym_label_type   * glbptr ;
  l_line_type      * lptr , * temp_lptr;
  err_type         * error ;
  char               message_buf[100] ;
  i = 0 ;
  /* Process global label list first */
  for ( glbptr = sym_glb_lab_head ; glbptr ; glbptr= glbptr->next ) {
     if ( glbptr->relative != '?' )   /* if its known then go to next one */
        continue ;
     e_message(0,23, NULL );
     i ++ ;
     error = e_error.errptr ;
     strcpy( message_buf , error->message ) ;
     strcpy( message_buf+strlen(error->message), glbptr->symbol );
     l_addline(l_neither, 0, message_buf , &lptr );
     e_delete_errors() ;
     }
  /* Process the local reference list. */
  for ( refptr = sym_local_ref_head ; refptr ; refptr= refptr->next ) {
     e_message(0,23, refptr->symbol );
     i ++ ;
     error = e_error.errptr ;
     strcpy( message_buf , error->message ) ;
     strcpy( message_buf+strlen(error->message), refptr->symbol );
     l_addline(l_neither, 0, message_buf , &lptr );
     l_addline(l_neither, 0, message_buf , &lptr );
     /* unlink the second redundant line */
     lptr->next->prev = lptr->prev ;
     lptr->prev->next = lptr->next ;
     /* link second line into listing where symbol was first referenced */
     temp_lptr = refptr->list->termptr->lineptr ;
     lptr->next = temp_lptr->next ;
     lptr->next->prev = lptr ;
     lptr->prev = temp_lptr ;
     temp_lptr->next = lptr ;
     e_delete_errors() ;
     }
  /* delete the local reference list */
  while ( refptr = sym_local_ref_head ) {
      while ( reflistptr = refptr->list ) {
         refptr->list = refptr->list->next ;
         free( reflistptr );
         }
      sym_local_ref_head = refptr->next ;
      free( refptr );
      }
  return i ;
} /* sym_process_unresolved_locals */
void sym_delete_all_tables( void )
{
 sym_operand_type * refptr ;
 sym_label_type   * labptr ;
 sym_ref_type     * reflistptr ;
 /* delete the global label list */
 while ( labptr = sym_glb_lab_head ) {
     sym_glb_lab_head = labptr->next ;
     free( labptr );
     }
 /* delete the local label list */
 while ( labptr = sym_local_lab_head ) {
     sym_local_lab_head = labptr->next ;
     free( labptr );
     }
 /* delete the local reference list */
 while ( refptr = sym_local_ref_head ) {
     while ( reflistptr = refptr->list ) {
        refptr->list = refptr->list->next ;
        free( reflistptr );
        }
     sym_local_ref_head = refptr->next ;
     free( refptr );
     }
 /* delete the external reference list */
 while ( refptr = sym_ext_ref_head ) {
     while ( reflistptr = refptr->list ) {
        refptr->list = refptr->list->next ;
        free( reflistptr );
        }
     sym_ext_ref_head = refptr->next ;
     free( refptr );
     }
} /* sym_delete_all_tables */
void sym_resolve_back( sym_label_type  *  symptr )
{
   unsigned int       i  ;
   sym_operand_type * temp, * prev ;
   sym_ref_type     * refptr , * refhead , * tempref ;
   p_sym_type       * sym ;
   am_term_type     * termptr ;
   int                warnings ;
   prev = NULL ;
   for ( temp = sym_local_ref_head ;
         temp && strncmp(temp->symbol,symptr->symbol, MAXSYMLEN ) ;
         prev = temp , temp = temp->next );
   if ( !temp ) return ;
                                        /* if there are back references */
   if ( prev )                        /* remove the operand node from list.*/
      prev->next = temp->next ;
   else
      sym_local_ref_head = temp->next ;
   refhead = temp->list ;
   free( temp ) ;
   refptr = refhead ;
   while ( refptr ) {
       termptr = refptr->termptr ;
       for ( sym = termptr->symptr ; sym ; sym= sym->next )
           if ( ! strncmp( sym->sym, symptr->symbol, MAXSYMLEN ) ) {
              /* if the symbol is in the symbol list of the term */
              /* then resolve it. */
              sym->val = symptr->val ;
              sym->sym[0] = symptr->relative ;
              sym->sym[1] = 1 ; /* for relative '*' put in count after it */
              }
       refptr = refptr->next ;
       }
   /* compress ref list so that only unique termlists are refered to. */
   refptr = refhead ;
   while ( refptr ) {
       termptr = refptr->termptr ;
       if ( termptr )
       switch ( termptr->class ) {
         case am_first_instr_term :
            /* get rid of any term pointers which refer to the */
            /* next term if its class is am_other_instr        */
            /* this works since instructions can have only 2 terms */
            if ( termptr->next )
               if ( termptr->next->class == am_other_instr_term ) {
                  for ( tempref = refhead; tempref ; tempref=tempref->next)
                      if ( tempref->termptr == termptr->next )
                         tempref->termptr = NULL ;
                  }
            break ;
         case am_other_instr_term :
            /* back up to first_instr_term and do same as in case above*/
            if ( termptr->prev )   /* there should always be a prev term */
               if ( termptr->prev->class == am_first_instr_term ) {
                  for ( tempref = refhead; tempref ; tempref=tempref->next)
                      if ( tempref->termptr == termptr->prev )
                         tempref->termptr = NULL ;
                  }
            refptr->termptr = termptr->prev ; /* set ptr to first term */
            break ;
         case am_data_term :
            /* resolve only one data term at a time. no compression */
            break;
         }
       refptr = refptr->next ;
```

(continued on page 106)

Listing Nine *(Listing continued, text begins on page 50.)*

```
      }
  /* resolve terms and dispose of reference list. */
  refptr = refhead ;
  while ( refptr ) {
     termptr = refptr->termptr ;
     tempref = refptr->next ;
     free(refptr);
     refptr = tempref ;
     if ( !termptr ) continue ;
     i=am_resolve_term( termptr,( termptr->class == am_data_term)?1:0);
     if ( !i ) {                              /* if all resolved */
        warnings = e_error.warnings ;
        am_backfill( termptr );
        if ( e_error.warnings > warnings )
           l_add_errors( termptr->lineptr );
        warnings = e_error.warnings ;
        am_remove_terms_from_list(&termptr) ;
        }
     }
} /* sym_resolve_back */
sym_label_type * sym_lookup_global( char  *  symbol )
{
  sym_label_type * temp ;

  for ( temp = sym_glb_lab_head ;
        temp && strncmp(temp->symbol,symbol, MAXSYMLEN ) ;
        temp = temp->next );
  return temp ;
} /* sym_lookup_global */
int  sym_add_global( char  *  symbol )
{
 sym_label_type * temp ;
 if ( sym_lookup_global( symbol ) ) {
    e_message(0,74,NULL) ;          /* WARNING symbol already in glb list.*/
    return 0 ;
    }
 if ( temp = ( sym_label_type * ) malloc ( sizeof(sym_label_type) ) ) {
    strncpy( temp->symbol, symbol, MAXSYMLEN );
    temp->relative = '?' ;
    temp->next = sym_glb_lab_head ;
    temp->val = 0L ;
    sym_glb_lab_head = temp ;
    return 0 ;
    }
 e_message(0,41,NULL) ; /* out of memory */
 return 41 ;
} /* sym_add_global */
sym_operand_type * sym_lookup_extern( char  *  symbol )
{
 sym_operand_type * temp ;
 for ( temp = sym_ext_ref_head ;
       temp && strncmp(temp->symbol,symbol, MAXSYMLEN ) ;
       temp = temp->next );
 return temp ;
} /* sym_lookup_ext */
int  sym_add_extern( char  *  symbol )
{
  sym_operand_type * temp ;
  if ( sym_lookup_extern( symbol ) ) {
     e_message(0,75,NULL) ;       /* WARNING symbol already in extern list.*/
     return 0 ;
     }
  if ( temp = ( sym_operand_type * ) malloc ( sizeof(sym_operand_type) ) ) {
     strncpy( temp->symbol, symbol, MAXSYMLEN );
     temp->next = sym_ext_ref_head ;
     temp->list = NULL ;
     sym_ext_ref_head = temp ;
     return 0 ;
     }
  e_message(0,41,NULL) ; /* out of memory */
  return 41 ;
} /* sym_add_extern */
sym_label_type * sym_lookup_local( char  *  symbol )
{
  sym_label_type * temp ;
  for ( temp = sym_local_lab_head ;
        temp && strncmp(temp->symbol,symbol, MAXSYMLEN ) ;
        temp = temp->next );
  return temp ;
} /* sym_lookup_local */
int  sym_add_label_symbol ( char          * symbol ,
                            unsigned long   val    ,
                            am_assem_type   relative )
{
  sym_label_type  * temp ;
  if ( sym_lookup_extern( symbol ) ) {
     e_message(0,17,NULL) ;       /* label symbol already in extern list.*/
     return 17;                   /* cannot resolve locally. */
     }
  if ( temp = sym_lookup_global( symbol ) ) {   /* in global list */
     if ( temp->relative == '?' ) {  /* not resolved yet. */
        temp->relative = ( relative == am_relative ) ? '*' : 0 ;
        temp->val = val ;
                                     /* resolve back references */
        sym_resolve_back( temp );
        return 0 ;
        }
     else {
        e_message(0,18,NULL) ;  /* symbol already resolved */
        return 18 ;
        }
     }
  if ( sym_lookup_local( symbol ) ) {
     e_message(0,18,NULL) ;       /* symbol already resolved in local list.*/
     return 18 ;
     }
  if ( temp = ( sym_label_type * ) malloc( sizeof(sym_label_type)) ) {
     strncpy( temp->symbol, symbol, MAXSYMLEN );
     temp->relative = ( relative == am_relative ) ? '*' : 0 ;
```

```
    temp->next = sym_local_lab_head ;
    temp->val  = val ;
    sym_local_lab_head = temp ;
                          /* resolve back references */
    sym_resolve_back( temp );
    return 0 ;
    }
  else {
    e_message(0,41,NULL) ;  /* out of memory */
    return 41 ;
    }
} /* sym_add_label_symbol */
int  sym_add_operand_symbol ( p_sym_type    * symptr  ,
                              am_term_type  * termptr ,
                              char            addref  )
{
  sym_operand_type * temp  ;
  sym_ref_type     * temp2 ;
  sym_label_type   * temp3 ;
  if ( temp = sym_lookup_extern( symptr->sym ) ) {  /* in extern list */
    if ( ! addref ) return 0 ;
    if ( temp2 = ( sym_ref_type * ) malloc( sizeof(sym_ref_type)) ) {
       temp2->termptr = termptr ;
       temp2->next = temp->list ;
       temp->list = temp2 ;
       return 0 ;
       }
    else {
      e_message(0,41,NULL) ;   /* not enough memory */
      return 41 ;
      }
    }
  if ( temp3 = sym_lookup_global( symptr->sym ) ) {   /* in global list */
    if ( temp3->relative == '?' )   /* not resolved yet. */
       if ( ! addref ) return 0 ;
       else return sym_addtolocaloplist( symptr, termptr ) ;
    else {
       symptr->val = temp3->val ;                  /* resolve the symbol */
       symptr->sym[0] = temp3->relative ;
       symptr->sym[1] = 1 ; /* set relative count to 1 */
       return 0 ;
       }
    }
  if ( temp3 = sym_lookup_local( symptr->sym ) ) {
    symptr->val = temp3->val ;                     /* resolve the symbol*/
    symptr->sym[0] = temp3->relative ;
    symptr->sym[1] = 1 ;   /* set relative count to 1 */
    return 0 ;
    }
  else
    if ( ! addref ) return 0 ;
    else return sym_addtolocaloplist( symptr, termptr ) ;
} /* sym_add_operand_symbol */
int  sym_addtolocaloplist ( p_sym_type    * symptr  ,
                            am_term_type  * termptr )
{
  sym_operand_type  * temp ;
  sym_ref_type      * temp2 ;
  for ( temp = sym_local_ref_head ;
        temp && strncmp(temp->symbol,symptr->sym, MAXSYMLEN ) ;
        temp = temp->next );
  if ( temp )
    if ( temp2 = ( sym_ref_type * ) malloc( sizeof(sym_ref_type)) ) {
       temp2->termptr = termptr ;
       temp2->next = temp->list ;
       temp->list = temp2 ;
       return 0 ;
       }
    else {
       e_message(0,41,NULL) ; /* not enough memory */
       return 41 ;
       }
  else
    if ( temp = ( sym_operand_type * ) malloc (sizeof(sym_operand_type)) )
       if ( temp2 = ( sym_ref_type * ) malloc( sizeof(sym_ref_type)) ) {
          strncpy( temp->symbol, symptr->sym, MAXSYMLEN );
          temp->next = sym_local_ref_head ;
          temp->list = temp2 ;
          sym_local_ref_head = temp ;
          temp2->termptr = termptr ;
          temp2->next = NULL ;
          return 0 ;
          }
       else {
          free(temp);
          e_message(0,41,NULL);  /* not enough memory */
          return 41 ;
          }
    else {
       e_message(0,41,NULL);  /* not enough memory */
       return 41 ;
       }
} /* sym_addtolocaloplist */
```

End Listings

Listing One *(Text begins on page 62.)*

```
program Edit;
uses OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpMemo, OpEditor;
var
  TE : TextEditor;
  FSize : LongInt;
  ExitCommand : Word;
  AllDone : Boolean;
begin
  if not TE.InitCustom(2, 4, 79, 24,          { Window coordinates }
                DefaultColorSet,              { ColorSet }
                DefWindowOptions or wBordered, { Win options }
                65521)                        { Buffer size }
  then
  begin
    WriteLn('Failed to init TextEditor. Status = ', InitStatus);
    Halt;
  end;
  { use built-in status and error handlers provided by OPMEMO }
  TE.SetStatusProc(MemoStatus);
  TE.SetErrorProc(MemoError);
  { Create and Read a text file }
  TE.ReadFile('AnyFile', FSize);
  AllDone := False;
  repeat
    TE.Process;
    ExitCommand := TE.GetLastCommand;
    case ExitCommand of
      ccSaveExit,           { Save and exit -- file already saved }
      ccAbandonFile,        { Abandon file }
      ccError :             { Fatal error }
        AllDone := True;
      {...user exit commands..}
    end;
  until AllDone;
  TE.Erase;
  TE.Done;
  ClrScr;
end.
```

End Listing One

Listing Two

```
program HelloWorld;
Uses HWMain, OpSwap;
begin
  HelloWorldMain;
end;

unit HWMain;
interface

uses
  OpInLine, OpSwap1;

procedure HelloWorldMain;

implementation

const
  HotKey = $080F { code for ALT-TAB }
  Swap1 = 'Hello1.SWP';
  Swap2 = 'Hello2.SWP';

  {$F+}
  procedure PopUpEntryPoint;
  begin
    Writeln('Hello World');
  end;
  {$F-}

  procedure HelloWorldMain;
  begin
    SetSwapMsgOn( not WillSwapUseEMS(ParagraphsToKeep) );
    { define the popup }
    if DefinePop(HotKey, PopUpEntryPoint, Ptr($SSeg,SPtr)) then
    begin
      Writeln('PopUp loaded, press <ALT><TAB> to activate.');
      { Make PopUp routines active. }
      PopUpsOn;
      { Try to go resident. }
      StayResSwap(ParagraphsToKeep, 0, Swap1, Swap2, True);
    end;

    { If we get here, report failure. }
    Writeln('Unable to go resident. ');
  end;

end.
```

End Listing Two

Listing Three

```
program CommandWindowExample;  {EXCMDWIN.PAS}
uses
  OpCrt, OpRoot, OpCmd, OpFrame, OpWindow;
const
  {Define a trivial KeySet of a few cursor commands}
  KeyMax  = 18;
  KeySet  : array[0..KeyMax] of Byte = (
  {length keys         command type   key sequence}
  3,      $00, $48,    ccUp,          {Up}
  3,      $00, $50,    ccDown,        {Down}
  3,      $00, $4B,    ccLeft,        {Left}
  3,      $00, $4D,    ccRight,       {Right}
  2,      $1B,         ccQuit);       {Esc}
type
  SampleWindow =
    object(CommandWindow)
      procedure Process; virtual;
    end;
var
  Commands : CommandProcessor;
  CmdWin : SampleWindow;
  Finished : Boolean;
  procedure SampleWindow.Process;
  begin
    repeat
      {Get a command}
      GetNextCommand;
      case GetLastCommand of
        ccUp :    WriteLn('ccUp');
        ccDown :  WriteLn('ccDown');
        ccLeft :  WriteLn('ccLeft');
        ccRight : WriteLn('ccRight');
        ccQuit :  WriteLn('ccQuit');
        ccChar :  WriteLn('ccChar: ', Char(Lo(GetLastKey)));
        else      WriteLn('ccNone');
      end;
    until (GetLastCommand = ccQuit) or (GetLastCommand = ccError);
  end;
begin
  {Make a small CommandProcessor}
  Commands.Init(@KeySet, KeyMax);
  {Make a bordered CommandWindow}
  if not CmdWin.InitCustom(30, 5, 50, 15,       {Window coordinates}
                DefaultColorSet,                {Color set}
                wBordered+wClear+wSaveContents, {Window options}
                Commands,                       {Command processor}
                ucNone)                         {Unit code}
  then begin
    WriteLn('Failed to init CommandWindow. Status = ', InitStatus);
    Halt;
  end;
  {Add headers and draw window}
  CmdWin.wFrame.AddHeader(' Command window ', heTC);
  CmdWin.wFrame.AddHeader(' <Esc> to Quit ', heBC);
  CmdWin.Draw;   p7 3
  {Get and process commands}
  Finished := False;
  repeat
    CmdWin.Process;
    case CmdWin.GetLastCommand of
      ccQuit : Finished := True;                        {Quit}
      ccError : begin                                   {Error}
                  WriteLn('Error: ', CmdWin.GetLastError);
                  Finished := True;
                end;
      ccUser0..ccUser55 : WriteLn('user command'); {Handle exit command}
    end;
  until Finished;
  {Clean up}
  CmdWin.Done;
  Commands.Done;
end.
```

End Listings

Listing One *(Text begins on page 70.)*

```
MODULE PCKermit;
(*************************************************************************)
(*                                                                       *)
(*                  PCKermit  --  by Brian R. Anderson                   *)
(*                        Copyright (c) 1990                             *)
(*                                                                       *)
(*  PCKermit is an implementation of the Kermit file transfer protocol   *)
(*  developed at Columbia University.  This (OS/2 PM) version is a       *)
(*  port from the DOS version of Kermit that I wrote two years ago.      *)
(*  My original DOS version appeared in the May 1989 issue of DDJ.       *)
(*                                                                       *)
(*  The current version includes emulation of the TVI950 Video Display   *)
(*  Terminal for interaction with IBM mainframes (through the IBM 7171).*)
(*                                                                       *)
(*************************************************************************)

FROM SYSTEM IMPORT
       ADR;

FROM OS2DEF IMPORT
       HAB, HWND, HPS, NULL, ULONG;

FROM PMWIN IMPORT
  MPFROM2SHORT, HMQ, QMSG, CS_SIZEREDRAW,  WS_VISIBLE, FS_ICON,
  FCF_TITLEBAR, FCF_SYSMENU, FCF_SIZEBORDER, FCF_MINMAX, FCF_ACCELTABLE,
  FCF_SHELLPOSITION, FCF_TASKLIST, FCF_MENU, FCF_ICON,
  SWP_MOVE, SWP_SIZE, SWP_MAXIMIZE,
  HWND_DESKTOP, FID_SYSMENU, SC_CLOSE, MIA_DISABLED, MM_SETITEMATTR,
  WinInitialize, WinCreateMsgQueue, WinGetMsg, WinDispatchMsg, WinSendMsg,
  WinRegisterClass, WinCreateStdWindow, WinDestroyWindow, WinWindowFromID,
  WinDestroyMsgQueue, WinTerminate, WinSetWindowText,
  WinSetWindowPos, WinQueryWindowPos;

   FROM KH IMPORT
      IDM_KERMIT;

   FROM Shell IMPORT
      Class, Title, Child, WindowProc, ChildWindowProc,
      FrameWindow, ClientWindow, SetPort, Pos;


   CONST
      QUEUE_SIZE = 1024;   (* Large message queue for async events *)

   VAR
      AnchorBlock : HAB;
      MessageQueue : HMQ;
      Message : QMSG;
      FrameFlags : ULONG;
      hsys : HWND;


BEGIN   (* main *)
   AnchorBlock := WinInitialize(0);

   IF AnchorBlock # 0 THEN
      MessageQueue := WinCreateMsgQueue (AnchorBlock, QUEUE_SIZE);

      IF MessageQueue # 0 THEN
         (* Register the parent window class *)
         WinRegisterClass (
             AnchorBlock,
             ADR (Class),
             WindowProc,
             CS_SIZEREDRAW, 0);

         (* Register a child window class *)
         WinRegisterClass (
             AnchorBlock,
             ADR (Child),
             ChildWindowProc,
             CS_SIZEREDRAW, 0);

         (* Create a standard window *)
         FrameFlags := FCF_TITLEBAR + FCF_MENU + FCF_MINMAX +
                       FCF_SYSMENU + FCF_SIZEBORDER + FCF_TASKLIST +
                       FCF_ICON + FCF_SHELLPOSITION + FCF_ACCELTABLE;

         FrameWindow := WinCreateStdWindow (
                  HWND_DESKTOP,          (* handle of the parent window *)
                  WS_VISIBLE + FS_ICON,  (* the window style *)
                  FrameFlags,            (* the window flags *)
                  ADR(Class),            (* the window class *)
                  NULL,                  (* the title bar text *)
                  WS_VISIBLE,            (* client window style *)
                  NULL,                  (* handle of resource module *)
                  IDM_KERMIT,            (* resource id *)
                  ClientWindow      (* returned client window handle *)
         );

         IF FrameWindow # 0 THEN
            (* Disable the CLOSE item on the system menu *)
            hsys := WinWindowFromID (FrameWindow, FID_SYSMENU);
            WinSendMsg (hsys, MM_SETITEMATTR,
               MPFROM2SHORT (SC_CLOSE, 1),
               MPFROM2SHORT (MIA_DISABLED, MIA_DISABLED));

           (* Expand Window to Nearly Full Size, And Display the Title *)
            WinQueryWindowPos (HWND_DESKTOP, ADR (Pos));
             WinSetWindowPos (FrameWindow, 0,
                Pos.x + 3, Pos.y + 3, Pos.cx - 6, Pos.cy - 6,
                SWP_MOVE + SWP_SIZE);
             WinSetWindowText (FrameWindow, ADR (Title));

             SetPort;   (* Try to initialize communications port *)

             WHILE WinGetMsg(AnchorBlock, Message, NULL, 0, 0) # 0 DO
                WinDispatchMsg(AnchorBlock, Message);
             END;

             WinDestroyWindow(FrameWindow);
          END;
          WinDestroyMsgQueue(MessageQueue);
       END;
       WinTerminate(AnchorBlock);
    END;
END PCKermit.
```

End Listing One

Listing Two

```
DEFINITION MODULE Shell;

   FROM OS2DEF IMPORT
      USHORT, HWND;

   FROM PMWIN IMPORT
      MPARAM, MRESULT, SWP;

   EXPORT QUALIFIED
      Class, Child, Title, FrameWindow, ClientWindow,
      ChildFrameWindow, ChildClientWindow, Pos, SetPort,
      WindowProc, ChildWindowProc;

   CONST
      Class = "PCKermit";
      Child ="Child";
      Title = "PCKermit -- Microcomputer to Mainframe Communications";


   VAR
      FrameWindow : HWND;
      ClientWindow : HWND;
      ChildFrameWindow : HWND;
      ChildClientWindow : HWND;
      Pos : SWP;   (* Screen Dimensions: position & size *)
      comport : CARDINAL;


   PROCEDURE SetPort;

   PROCEDURE WindowProc ['WindowProc'] (
      hwnd : HWND;
      msg  : USHORT;
      mp1  : MPARAM;
      mp2  : MPARAM) : MRESULT [LONG, LOADDS];

   PROCEDURE ChildWindowProc ['ChildWindowProc'] (
      hwnd : HWND;
      msg  : USHORT;
      mp1  : MPARAM;
      mp2  : MPARAM) : MRESULT [LONG, LOADDS];

END Shell.
```

End Listing Two

Listing Three

```
DEFINITION MODULE Term;   (* TVI950 Terminal Emulation For Kermit *)

   EXPORT QUALIFIED
      WM_TERM, WM_TERMQUIT,
      Dir, TermThrProc, InitTerm, PutKbdChar, PutPortChar;

   CONST
      WM_TERM = 4000H;
      WM_TERMQUIT = 4001H;


   PROCEDURE Dir (path : ARRAY OF CHAR);
   (* Displays a directory *)

   PROCEDURE TermThrProc;
   (* Thread to get characters from port, put into buffer, send message.*)

   PROCEDURE InitTerm;
   (* Clear Screen, Home Cursor, Get Ready For Terminal Emulation *)

   PROCEDURE PutKbdChar (ch1, ch2 : CHAR);
   (* Process a character received from the keyboard*)

   PROCEDURE PutPortChar (ch : CHAR);
   (* Process a character received from the port *)

END Term.
```

End Listing Three

Listing Four

```
DEFINITION MODULE Screen;
(* Module to perform "low level" screen functions (via AVIO) *)

   FROM PMAVIO IMPORT
      HVPS;
```

(continued on page 110)

Listing Four *(Listing continued, text begins on page 70.)*

```
199:    EXPORT QUALIFIED
200:       NORMAL, HIGHLIGHT, REVERSE, attribute, ColorSet, hvps,
201:       White, Green, Amber, Color1, Color2,
202:       ClrScr, ClrEol, GotoXY, GetXY,
203:       Right, Left, Up, Down, Write, WriteLn, WriteString,
204:       WriteInt, WriteHex, WriteAtt;
205:
206:
207:    VAR
208:       NORMAL : CARDINAL;
209:       HIGHLIGHT : CARDINAL;
210:       REVERSE : CARDINAL;
211:       attribute : CARDINAL;
212:       ColorSet : CARDINAL;
213:       hvps : HVPS;   (* presentation space used by screen module *)
214:
215:
216:    PROCEDURE White;
217:    (* Sets up colors: Monochrome White *)
218:
219:    PROCEDURE Green;
220:    (* Sets up colors: Monochrome Green *)
221:
222:    PROCEDURE Amber;
223:    (* Sets up colors: Monochrome Amber *)
224:
225:    PROCEDURE Color1;
226:    (* Sets up colors: Blue, Red, Green *)
227:
228:    PROCEDURE Color2;
229:    (* Sets up colors: Green, Magenta, Cyan *)
230:
231:    PROCEDURE ClrScr;
232:    (* Clear the screen, and home the cursor *)
233:
234:    PROCEDURE ClrEol;
235:    (* clear from the current cursor position to the end of the line *)
236:
237:    PROCEDURE Right;
238:    (* move cursor to the right *)
239:
240:    PROCEDURE Left;
241:    (* move cursor to the left *)
242:
243:    PROCEDURE Up;
244:    (* move cursor up *)
245:
246:    PROCEDURE Down;
247:    (* move cursor down *)
248:
249:    PROCEDURE GotoXY (col, row : CARDINAL);
250:    (* position cursor at column, row *)
251:
252:    PROCEDURE GetXY (VAR col, row : CARDINAL);
253:    (* determine current cursor position *)
254:
255:    PROCEDURE Write (c : CHAR);
256:    (* Write a Character, Teletype Mode *)
257:
258:    PROCEDURE WriteString (str : ARRAY OF CHAR);
259:    (* Write String, Teletype Mode *)
260:
261:    PROCEDURE WriteInt (n : INTEGER; s : CARDINAL);
262:    (* Write Integer, Teletype Mode *)
263:
264:    PROCEDURE WriteHex (n, s : CARDINAL);
265:    (* Write a Hexadecimal Number, Teletype Mode *)
266:
267:    PROCEDURE WriteLn;
268:    (* Write <cr> <lf>, Teletype Mode *)
269:
270:    PROCEDURE WriteAtt (c : CHAR);
271:    (* write character and attribute at cursor position *)
272:
273: END Screen.
274:
275:
```

End Listing Four

Listing Five

```
276: DEFINITION MODULE PAD;   (* Packet Assembler/Disassembler for Kermit *)
277:
278:    FROM PMWIN IMPORT
279:       MPARAM;
280:
281: EXPORT QUALIFIED
282:   WM_PAD, PAD_Quit, PAD_Error, PacketType, yourNPAD, yourPADC, yourEOL,
283:   Aborted, sFname, Send, Receive, DoPADMsg;
284:
285:    CONST
286:       WM_PAD = 5000H;
287:       PAD_Quit = 0;
288:       PAD_Error = 20;
289:
290:    TYPE
291:       (* PacketType used in both PAD and DataLink modules *)
292:       PacketType = ARRAY [1..100] OF CHAR;
293:
294:    VAR
295:  (* yourNPAD, yourPADC, and yourEOL used in both PAD and DataLink *)
296:       yourNPAD : CARDINAL;    (* number of padding characters *)
297:       yourPADC : CHAR;        (* padding characters *)
298:       yourEOL  : CHAR;        (* End Of Line -- terminator *)
299:       sFname : ARRAY [0..20] OF CHAR;
300:       Aborted : BOOLEAN;
301:
302:    PROCEDURE Send;
303:    (* Sends a file after prompting for filename *)
304:
305:    PROCEDURE Receive;
306:    (* Receives a file (or files) *)
307:
308:    PROCEDURE DoPADMsg (mp1, mp2 : MPARAM);
309:    (* Output messages for Packet Assembler/Disassembler *)
310:
311: END PAD.
312:
```

End Listing Five

Listing Six

```
313: DEFINITION MODULE DataLink;   (* Sends/Receives Packets for PCKermit *)
314:
315:    FROM PMWIN IMPORT
316:       MPARAM;
317:
318:    FROM PAD IMPORT
319:       PacketType;
320:
321:    EXPORT QUALIFIED
322:       WM_DL, FlushUART, SendPacket, ReceivePacket, DoDLMsg;
323:
324:    CONST
325:       WM_DL = 6000H;
326:
327:    PROCEDURE FlushUART;
328:    (* ensure no characters left in UART holding registers *)
329:
330:    PROCEDURE SendPacket (s : PacketType);
331:    (* Adds SOH and CheckSum to packet *)
332:
333:    PROCEDURE ReceivePacket (VAR r : PacketType) : BOOLEAN;
334:    (* strips SOH and checksum -- returns status: TRUE = good packet       *)
335:    (* received;  FALSE = timed out waiting for packet or checksum error  *)
336:
337:    PROCEDURE DoDLMsg (mp1, mp2 : MPARAM);
338:    (* Process DataLink Messages *)
339:
340: END DataLink.
341:
```

End Listing Six

Listing Seven

```
342: (*************************************************************)
343: (*                                                           *)
344: (*               Copyright (C) 1988, 1989                    *)
345: (*                by Stony Brook Software                    *)
346: (*                                                           *)
347: (*                  All rights reserved.                     *)
348: (*                                                           *)
349: (*************************************************************)
```

```
350:
351:  DEFINITION MODULE CommPort;
352:
353:     TYPE
354:        CommStatus = (
355:                Success,
356:                InvalidPort,
357:                InvalidParameter,
358:                AlreadyReceiving,
359:                NotReceiving,
360:                NoCharacter,
361:                FramingError,
362:                OverrunError,
363:                ParityError,
364:                BufferOverflow,
365:                TimeOut
366:        );
367:
368:        BaudRate = (
369:                Baud110,
370:                Baud150,
371:                Baud300,
372:                Baud600,
373:                Baud1200,
374:                Baud2400,
375:                Baud4800,
376:                Baud9600,
377:                Baud19200
378:        );
379:
380:        DataBits = [7..8];
381:        StopBits = [1..2];
382:        Parity = (Even, Odd, None);
383:
384:
385:     PROCEDURE InitPort(port : CARDINAL; speed : BaudRate; data : DataBits;
386:                        stop : StopBits; check : Parity) : CommStatus;
387:
388:     PROCEDURE StartReceiving(port, bufsize : CARDINAL) : CommStatus;
389:
390:     PROCEDURE StopReceiving(port : CARDINAL) : CommStatus;
391:
392:     PROCEDURE GetChar(port : CARDINAL; VAR ch : CHAR) : CommStatus;
393:
394:     PROCEDURE SendChar(port : CARDINAL; ch : CHAR; modem :
395:                                                BOOLEAN) : CommStatus;
396:  END CommPort.
397:
```

End Listing Seven

Listing Eight

```
398:  DEFINITION MODULE Files;   (* File I/O for Kermit *)
399:
400:     FROM FileSystem IMPORT
401:        File;
402:
403:     EXPORT QUALIFIED
404:        Status, FileType, Open, Create, CloseFile, Get, Put, DoWrite;
405:
406:     TYPE
407:        Status = (Done, Error, EOF);
408:        FileType = (Input, Output);
409:
410:     PROCEDURE Open (VAR f : File; name : ARRAY OF CHAR) : Status;
411:     (* opens an existing file for reading, returns status *)
412:
413:     PROCEDURE Create (VAR f : File; name : ARRAY OF CHAR) : Status;
414:     (* creates a new file for writing, returns status *)
415:
416:     PROCEDURE CloseFile (VAR f : File; Which : FileType) : Status;
417:     (* closes a file after reading or writing *)
418:
419:     PROCEDURE Get (VAR f : File; VAR ch : CHAR) : Status;
420:     (* Reads one character from the file, returns status *)
421:
422:     PROCEDURE Put (ch : CHAR);
423:     (* Writes one character to the file buffer *)
424:
425:     PROCEDURE DoWrite (VAR f : File) : Status;
426:     (* Writes buffer to disk only if nearly full *)
427:
428:  END Files.
429:
```

End Listing Eight

Listing Nine

```
430:  IMPLEMENTATION MODULE Shell;
431:
432:     FROM SYSTEM IMPORT
433:        ADDRESS, ADR;
434:
435:     IMPORT ASCII;
436:
437:     FROM OS2DEF IMPORT
438:        LOWORD, HIWORD, HWND, HDC, HPS, RECTL, USHORT, NULL, ULONG;
439:
440:     FROM Term IMPORT
441:        WM_TERM, WM_TERMQUIT,
442:        Dir, TermThrProc, InitTerm, PutKbdChar, PutPortChar;
443:
444:     FROM PAD IMPORT
445:        WM_PAD, PAD_Quit, PAD_Error, DoPADMsg, Aborted, sFname, Send,
                                                                  Receive;
446:
```

(continued on page 112)

Listing Nine *(Listing continued, text begins on page 70.)*

```
447:    FROM DataLink IMPORT
448:       WM_DL, DoDLMsg;
449:
450:    FROM Screen IMPORT
451:       hvps, ColorSet, White, Green, Amber, Color1, Color2, ClrScr,
                                                              WriteLn;
452:
453:    FROM DosCalls IMPORT
454:       DosCreateThread, DosSuspendThread, DosResumeThread, DosSleep;
455:
456:    FROM PMAVIO IMPORT
457:       VioCreatePS, VioAssociate, VioDestroyPS, VioShowPS,
                                                  WinDefAVioWindowProc,
458:       FORMAT_CGA, HVPS;
459:
460:    FROM PMWIN IMPORT
461:       MPARAM, MRESULT, SWP, PSWP,
462:       WS_VISIBLE, FCF_TITLEBAR, FCF_SIZEBORDER, FCF_SHELLPOSITION,
463:       WM_SYSCOMMAND, WM_MINMAXFRAME, SWP_MINIMIZE, HWND_DESKTOP,
464:       WM_PAINT, WM_QUIT, WM_COMMAND, WM_INITDLG, WM_CONTROL, WM_HELP,
465:       WM_INITMENU, WM_SIZE, WM_DESTROY, WM_CREATE, WM_CHAR,
466:       BM_SETCHECK, MBID_OK, MB_OK, MB_OKCANCEL,
467:       KC_CHAR, KC_CTRL, KC_VIRTUALKEY, KC_KEYUP,
468:       SWP_SIZE, SWP_MOVE, SWP_MAXIMIZE, SWP_RESTORE,
469:       MB_ICONQUESTION, MB_ICONASTERISK, MB_ICONEXCLAMATION,
470:       FID_MENU, MM_SETITEMATTR, MM_QUERYITEMATTR,
471:       MIA_DISABLED, MIA_CHECKED, MPFROM2SHORT,
472:       WinCreateStdWindow, WinDestroyWindow,
473:       WinOpenWindowDC, WinSendMsg, WinQueryDlgItemText, WinInvalidateRect,
474:       WinDefWindowProc, WinBeginPaint, WinEndPaint, WinQueryWindowRect,
475:       WinSetWindowText, WinSetFocus, WinDlgBox, WinDefDlgProc,
                                                        WinDismissDlg,
476:       WinMessageBox, WinPostMsg, WinWindowFromID, WinSendDlgItemMsg,
477:       WinSetWindowPos, WinSetActiveWindow;
478:
479:    FROM PMGPI IMPORT
480:       GpiErase;
481:
482:    FROM KH IMPORT
483:       IDM_KERMIT, IDM_FILE, IDM_OPTIONS, IDM_SENDFN, ID_SENDFN,
484:       IDM_DIR, IDM_CONNECT, IDM_SEND, IDM_REC, IDM_DIRPATH, ID_DIRPATH,
485:       IDM_DIREND, IDM_QUIT, IDM_ABOUT, IDM_HELPMENU, IDM_TERMHELP,
486:       IDM_COMPORT, IDM_BAUDRATE, IDM_DATABITS, IDM_STOPBITS, IDM_PARITY,
487:       COM_OFF, ID_COM1, ID_COM2, PARITY_OFF, ID_EVEN, ID_ODD, ID_NONE,
488:       DATA_OFF, ID_DATA7, ID_DATA8, STOP_OFF, ID_STOP1, ID_STOP2,
489:       BAUD_OFF, ID_B110, ID_B150, ID_B300, ID_B600, ID_B1200, ID_B2400,
490:       ID_B4800, ID_B9600, ID_B19K2,
491:       IDM_COLORS, IDM_WHITE, IDM_GREEN, IDM_AMBER, IDM_C1, IDM_C2;
492:
493:    FROM CommPort IMPORT
494:       CommStatus, BaudRate, DataBits, StopBits, Parity, InitPort,
495:       StartReceiving, StopReceiving;
496:
497:    FROM Strings IMPORT
498:       Assign, Append, AppendChar;
499:
500:
501:    CONST
502:       WM_SETMAX = 7000H;
503:       WM_SETFULL = 7001H;
504:       WM_SETRESTORE = 7002H;
505:       NONE = 0;   (* no port yet initialized *)
506:       STKSIZE = 4096;
507:       BUFSIZE = 4096;   (* Port receive buffers: room for two
                                                        full screens *)
508:       PortError = "Port Is Already In Use -- EXIT? (Cancel Trys
                                                        Another Port)";
509:       ESC = 33C;
510:
511:
512:    VAR
513:       FrameFlags : ULONG;
514:       TermStack : ARRAY [1..STKSIZE] OF CHAR;
515:       Stack : ARRAY [1..STKSIZE] OF CHAR;
516:       TermThr : CARDINAL;
517:       Thr : CARDINAL;
518:       hdc : HDC;
519:       frame_hvps, child_hvps : HVPS;
520:       TermMode : BOOLEAN;
521:       Path : ARRAY [0..60] OF CHAR;
522:       Banner : ARRAY [0..40] OF CHAR;
523:       PrevComPort : CARDINAL;
524:       Settings : ARRAY [0..1] OF RECORD
525:                                    baudrate : CARDINAL;
526:                                    databits : CARDINAL;
527:                                    parity : CARDINAL;
528:                                    stopbits : CARDINAL;
529:                                  END;
530:
531:    PROCEDURE SetFull;
532:    (* Changes window to full size *)
533:       BEGIN
534:          WinSetWindowPos (FrameWindow, 0,
535:             Pos.x + 3, Pos.y + 3, Pos.cx - 6, Pos.cy - 6,
536:             SWP_MOVE + SWP_SIZE);
537:       END SetFull;
538:
539:
540:    PROCEDURE SetRestore;
541:    (* Changes window to full size FROM maximized *)
542:       BEGIN
543:          WinSetWindowPos (FrameWindow, 0,
544:             Pos.x + 3, Pos.y + 3, Pos.cx - 6, Pos.cy - 6,
545:             SWP_MOVE + SWP_SIZE + SWP_RESTORE);
546:       END SetRestore;
547:
548:
549:    PROCEDURE SetMax;
550:    (* Changes window to maximized *)
551:       BEGIN
552:          WinSetWindowPos (FrameWindow, 0,
553:             Pos.x + 3, Pos.y + 3, Pos.cx - 6, Pos.cy - 6,
554:             SWP_MOVE + SWP_SIZE + SWP_MAXIMIZE);
555:       END SetMax;
556:
557:
558:    PROCEDURE SetBanner;
559:    (* Displays Abbreviated Program Title + Port Settings in
                                                        Title Bar *)
560:
561:       CONST
562:          PortName : ARRAY [0..1] OF ARRAY [0..5] OF CHAR =
563:             [["COM1:", 0C], ["COM2:", 0C]];
564:          BaudName : ARRAY [0..8] OF ARRAY [0..5] OF CHAR =
565:             [["110", 0C], ["150", 0C], ["300", 0C],
566:              ["600", 0C], ["1200", 0C], ["2400", 0C],
567:              ["4800", 0C], ["9600", 0C], ["19200", 0C]];
568:          ParityName : ARRAY [0..2] OF CHAR = ['E', 'O', 'N'];
569:
570:       BEGIN
571:          WITH Settings[comport - COM_OFF] DO
572:             Assign (Class, Banner);
573:             Append (Banner, " -- ");
574:             Append (Banner, PortName[comport - COM_OFF]);
575:             Append (Banner, BaudName[baudrate - BAUD_OFF]);
576:             AppendChar (Banner, ',');
577:             AppendChar (Banner, ParityName[parity - PARITY_OFF]);
578:             AppendChar (Banner, ',');
579:             AppendChar (Banner, CHR ((databits - DATA_OFF) + 30H));
580:             AppendChar (Banner, ',');
581:             AppendChar (Banner, CHR ((stopbits - STOP_OFF) + 30H));
582:             WinSetWindowText (FrameWindow, ADR (Banner));
583:          END;
584:       END SetBanner;
585:
586:
587:    PROCEDURE SetPort;
588:    (* Sets The Communications Parameters Chosen By User *)
589:
590:       VAR
591:          status : CommStatus;
592:          rc : USHORT;
593:
594:       BEGIN
595:          IF PrevComPort # NONE THEN
596:             StopReceiving (PrevComPort - COM_OFF);
597:          END;
598:
599:          WITH Settings[comport - COM_OFF] DO
600:             status := InitPort (
601:                comport - COM_OFF,
602:                BaudRate (baudrate - BAUD_OFF),
603:                DataBits (databits - DATA_OFF),
604:                StopBits (stopbits - STOP_OFF),
605:                Parity (parity - PARITY_OFF),
606:             );
607:          END;
608:
609:          IF status = Success THEN
610:             StartReceiving (comport - COM_OFF, BUFSIZE);
611:             PrevComPort := comport;
612:          ELSE
613:             rc := WinMessageBox (HWND_DESKTOP, FrameWindow,
                                                        ADR (PortError),
614:                                  0, 0, MB_OKCANCEL +
                                                        MB-IONEXCLAMATION);
615:             IF rc = MBID_OK THEN
616:                WinPostMsg (FrameWindow, WM_QUIT, 0, 0);
617:             ELSE   (* try the other port *)
618:                IF comport = ID_COM1 THEN
619:                   comport := ID_COM2;
620:                ELSE
621:                   comport := ID_COM1;
622:                END;
623:                SetPort;   (* recursive call for retry *)
624:             END;
625:          END;
626:          SetBanner;
627:       END SetPort;
628:
629:
630:    PROCEDURE MakeChild (msg : ARRAY OF CHAR);
631:    (* Creates a child window for use by send or receive threads *)
632:
633:       VAR
634:          c_hdc : HDC;
635:
636:       BEGIN
637:          WinPostMsg (FrameWindow, WM_SETFULL, 0, 0);
638:
639:          Disable (IDM_CONNECT);
640:          Disable (IDM_SEND);
641:          Disable (IDM_REC);
642:          Disable (IDM_DIR);
643:          Disable (IDM_OPTIONS);
644:          Disable (IDM_COLORS);
645:
646:          (* Create a client window *)
647:          FrameFlags := FCF_TITLEBAR + FCF_SIZEBORDER;
648:
649:          ChildFrameWindow := WinCreateStdWindow (
650:                 ClientWindow,        (* handle of the parent window *)
651:                  WS_VISIBLE, (* the window style *)
652:                  FrameFlags,          (* the window flags *)
653:                  ADR(Child),          (* the window class *)
654:                  NULL,                (* the title bar text *)
655:                  WS_VISIBLE,          (* client window style *)
656:                  NULL,                (* handle of resource module *)
657:                  IDM_KERMIT,          (* resource id *)
658:                  ChildClientWindow    (* returned   window handle *)
659:          );
```

```
660:
661:            WinSetWindowPos (ChildFrameWindow, 0,
662:               Pos.cx DIV 4, Pos.cy DIV 4,
663:               Pos.cx DIV 2, Pos.cy DIV 2 - 3,
664:               SWP_MOVE + SWP_SIZE);
665:
666:            WinSetWindowText (ChildFrameWindow, ADR (msg));
667:
668:            WinSetActiveWindow (HWND_DESKTOP, ChildFrameWindow);
669:
670:            c_hdc := WinOpenWindowDC (ChildClientWindow);
671:            hvps := child_hvps;
672:            VioAssociate (c_hdc, hvps);
673:            ClrScr;  (* clear the hvio window *)
674:         END MakeChild;
675:
676:
677:      PROCEDURE Disable (item : USHORT);
678:      (* Disables and "GREYS" a menu item *)
679:
680:         VAR
681:            h : HWND;
682:
683:         BEGIN
684:            h := WinWindowFromID (FrameWindow, FID_MENU);
685:            WinSendMsg (h, MM_SETITEMATTR,
686:               MPFROM2SHORT (item, 1),
687:               MPFROM2SHORT (MIA_DISABLED, MIA_DISABLED));
688:         END Disable;
689:
690:
691:      PROCEDURE Enable (item : USHORT);
692:      (* Enables a menu item *)
693:
694:         VAR
695:            h : HWND;
696:            atr : USHORT;
697:
698:         BEGIN
699:            h := WinWindowFromID (FrameWindow, FID_MENU);
700:            atr := USHORT (WinSendMsg (h, MM_QUERYITEMATTR,
701:                           MPFROM2SHORT (item, 1),
702:                           MPFROM2SHORT (MIA_DISABLED, MIA_DISABLED)));
703:            atr := USHORT (BITSET (atr) * (BITSET (MIA_DISABLED)
                                                    / BITSET (-1)));
704:            WinSendMsg (h, MM_SETITEMATTR,
705:               MPFROM2SHORT (item, 1),
706:               MPFROM2SHORT (MIA_DISABLED, atr));
707:         END Enable;
708:
709:
710:      PROCEDURE Check (item : USHORT);
711:      (* Checks a menu item -- indicates that it is selected *)
712:
713:         VAR
```

```
714:            h : HWND;
715:
716:         BEGIN
717:            h := WinWindowFromID (FrameWindow, FID_MENU);
718:            WinSendMsg (h, MM_SETITEMATTR,
719:               MPFROM2SHORT (item, 1),
720:               MPFROM2SHORT (MIA_CHECKED, MIA_CHECKED));
721:         END Check;
722:
723:
724:      PROCEDURE UnCheck (item : USHORT);
725:      (* Remove check from a menu item *)
726:
727:         VAR
728:            h : HWND;
729:            atr : USHORT;
730:
731:         BEGIN
732:            h := WinWindowFromID (FrameWindow, FID_MENU);
733:            atr := USHORT (WinSendMsg (h, MM_QUERYITEMATTR,
734:                           MPFROM2SHORT (item, 1),
735:                           MPFROM2SHORT (MIA_CHECKED, MIA_CHECKED)));
736:            atr := USHORT (BITSET (atr) * (BITSET (MIA_CHECKED) /
                                                    BITSET (-1)));
737:            WinSendMsg (h, MM_SETITEMATTR,
738:               MPFROM2SHORT (item, 1),
739:               MPFROM2SHORT (MIA_CHECKED, atr));
740:         END UnCheck;
741:
742:
743:      PROCEDURE DoMenu (hwnd : HWND; item : MPARAM);
744:      (* Processes Most Menu Interactions *)
745:
746:         VAR
747:            rcl : RECTL;
748:            rc : USHORT;
749:
750:         BEGIN
751:            CASE  LOWORD (item) OF
752:               IDM_DIR:
753:                  SetFull;
754:                  WinQueryWindowRect (hwnd, rcl);
755:                  WinDlgBox (HWND_DESKTOP, hwnd, PathDlgProc, 0,
                                                    IDM_DIRPATH, 0);
756:                  hvps := frame_hvps;
757:                  VioAssociate (hdc, hvps);
758:                  Dir (Path);
759:                  WinDlgBox (HWND_DESKTOP, hwnd, DirEndDlgProc, 0,
                                                    IDM_DIREND, 0);
760:                  VioAssociate (0, hvps);
761:                  WinInvalidateRect (hwnd, ADR (rcl), 0);
762:               | IDM_CONNECT:
763:                  TermMode := TRUE;
```

(continued on page 114)

(Listing continued, text begins on page 70.)

```
764:                Disable (IDM_CONNECT);
765:                Disable (IDM_SEND);
766:                Disable (IDM_REC);
767:                Disable (IDM_DIR);
768:                Disable (IDM_OPTIONS);
769:                Disable (IDM_COLORS);
770:                (* MAXIMIZE Window -- Required for Terminal Emulation *)
771:                SetMax;
772:                hvps := frame_hvps;
773:                VioAssociate (hdc, hvps);
774:                DosResumeThread (TermThr);
775:                InitTerm;
776:             | IDM_SEND:
777:                WinDlgBox (HWND_DESKTOP, hwnd, SendFNDlgProc, 0,
                                                                IDM_SENDFN, 0);
778:                MakeChild ("Send a File");
779:                DosCreateThread (Send, Thr, ADR (Stack[STKSIZE]));
780:             | IDM_REC:
781:                MakeChild ("Receive a File");
782:                DosCreateThread (Receive, Thr, ADR (Stack[STKSIZE]));
783:             | IDM_QUIT:
784:                rc := WinMessageBox (HWND_DESKTOP, ClientWindow,
785:                        ADR ("Do You Really Want To EXIT PCKermit?"),
786:                        ADR ("End Session"), 0, MB_OKCANCEL +
                                                              MB_ICONQUESTION);
787:                IF rc = MBID_OK THEN
788:                   StopReceiving (comport - COM_OFF);
789:                   WinPostMsg (hwnd, WM_QUIT, 0, 0);
790:                END;
791:             | IDM_COMPORT:
792:                WinDlgBox (HWND_DESKTOP, hwnd, ComDlgProc, 0,
                                                               IDM_COMPORT, 0);
793:                SetPort;
794:             | IDM_BAUDRATE:
795:                WinDlgBox (HWND_DESKTOP, hwnd, BaudDlgProc, 0,
                                                              IDM_BAUDRATE, 0);
796:                SetPort;
797:             | IDM_DATABITS:
798:                WinDlgBox (HWND_DESKTOP, hwnd, DataDlgProc, 0,
                                                              IDM_DATABITS, 0);
799:                SetPort;
800:             | IDM_STOPBITS:
801:                WinDlgBox (HWND_DESKTOP, hwnd, StopDlgProc, 0,
                                                              IDM_STOPBITS, 0);
802:                SetPort;
803:             | IDM_PARITY:
804:                WinDlgBox (HWND_DESKTOP, hwnd, ParityDlgProc, 0,
                                                               IDM_PARITY, 0);
805:                SetPort;
806:             | IDM_WHITE:
807:                UnCheck (ColorSet);
808:                ColorSet := IDM_WHITE;
809:                Check (ColorSet);
810:                White;
811:             | IDM_GREEN:
812:                UnCheck (ColorSet);
813:                ColorSet := IDM_GREEN;
814:                Check (ColorSet);
815:                Green;
816:             | IDM_AMBER:
817:                UnCheck (ColorSet);
818:                ColorSet := IDM_AMBER;
819:                Check (ColorSet);
820:                Amber;
821:             | IDM_C1:
822:                UnCheck (ColorSet);
823:                ColorSet := IDM_C1;
824:                Check (ColorSet);
825:                Color1;
826:             | IDM_C2:
827:                UnCheck (ColorSet);
828:                ColorSet := IDM_C2;
829:                Check (ColorSet);
830:                Color2;
831:             | IDM_ABOUT:
832:                WinDlgBox (HWND_DESKTOP, hwnd, AboutDlgProc, 0,
                                                               IDM_ABOUT, 0);
833:          ELSE
834:             (* Don't do anything... *)
835:          END;
836:       END DoMenu;
837:
838:
839:    PROCEDURE ComDlgProc ['ComDlgProc'] (
840:    (* Process Dialog Box for choosing COM1/COM2 *)
841:          hwnd  : HWND;
842:          msg   : USHORT;
843:          mp1   : MPARAM;
844:          mp2   : MPARAM) : MRESULT [LONG, LOADDS];
845:       BEGIN
846:          CASE msg OF
847:             WM_INITDLG:
848:                WinSendDlgItemMsg (hwnd, comport, BM_SETCHECK, 1, 0);
849:                WinSetFocus (HWND_DESKTOP, WinWindowFromID
                                                          (hwnd, comport));
850:                RETURN 1;
851:          | WM_CONTROL:
852:                comport := LOWORD (mp1);
853:                RETURN 0;
854:          | WM_COMMAND:
855:                WinDismissDlg (hwnd, 1);
856:                RETURN 0;
857:          ELSE
858:             RETURN WinDefDlgProc (hwnd, msg, mp1, mp2);
859:          END;
860:       END ComDlgProc;
861:
862:
863:    PROCEDURE BaudDlgProc ['BaudDlgProc'] (
```

```
(* Process Dialog Box for choosing Baud Rate *)
     hwnd  : HWND;
     msg   : USHORT;
     mp1   : MPARAM;
     mp2   : MPARAM) : MRESULT [LONG, LOADDS];
  BEGIN
     WITH Settings[comport - COM_OFF] DO
        CASE msg OF
           WM_INITDLG:
              WinSendDlgItemMsg (hwnd, baudrate, BM_SETCHECK, 1, 0);
              WinSetFocus (HWND_DESKTOP, WinWindowFromID
                                                   (hwnd, baudrate));
              RETURN 1;
        |  WM_CONTROL:
              baudrate := LOWORD (mp1);
              RETURN 0;
        |  WM_COMMAND:
              WinDismissDlg (hwnd, 1);
              RETURN 0;
        ELSE
           RETURN WinDefDlgProc (hwnd, msg, mp1, mp2);
        END;
     END;
  END BaudDlgProc;


PROCEDURE DataDlgProc ['DataDlgProc'] (
(* Process Dialog Box for choosing 7 or 8 data bits *)
     hwnd  : HWND;
     msg   : USHORT;
     mp1   : MPARAM;
     mp2   : MPARAM) : MRESULT [LONG, LOADDS];
  BEGIN
     WITH Settings[comport - COM_OFF] DO
        CASE msg OF
           WM_INITDLG:
              WinSendDlgItemMsg (hwnd, databits, BM_SETCHECK, 1, 0);
              WinSetFocus (HWND_DESKTOP, WinWindowFromID
                                                   (hwnd, databits));
              RETURN 1;
        |  WM_CONTROL:
              databits := LOWORD (mp1);
              RETURN 0;
        |  WM_COMMAND:
              WinDismissDlg (hwnd, 1);
              RETURN 0;
        ELSE
           RETURN WinDefDlgProc (hwnd, msg, mp1, mp2);
        END;
     END;
  END DataDlgProc;


PROCEDURE StopDlgProc ['StopDlgProc'] (
(* Process Dialog Box for choosing 1 or 2 stop bits *)
     hwnd  : HWND;
     msg   : USHORT;
     mp1   : MPARAM;
     mp2   : MPARAM) : MRESULT [LONG, LOADDS];
  BEGIN
     WITH Settings[comport - COM_OFF] DO
        CASE msg OF
           WM_INITDLG:
              WinSendDlgItemMsg (hwnd, stopbits, BM_SETCHECK, 1, 0);
              WinSetFocus (HWND_DESKTOP, WinWindowFromID
                                                   (hwnd, stopbits));
              RETURN 1;
        |  WM_CONTROL:
              stopbits := LOWORD (mp1);
              RETURN 0;
        |  WM_COMMAND:
              WinDismissDlg (hwnd, 1);
              RETURN 0;
        ELSE
           RETURN WinDefDlgProc (hwnd, msg, mp1, mp2);
        END;
     END;
  END StopDlgProc;


PROCEDURE ParityDlgProc ['ParityDlgProc'] (
(* Process Dialog Box for choosing odd, even, or no parity *)
     hwnd  : HWND;
     msg   : USHORT;
     mp1   : MPARAM;
     mp2   : MPARAM) : MRESULT [LONG, LOADDS];
  BEGIN
     WITH Settings[comport - COM_OFF] DO
        CASE msg OF
           WM_INITDLG:
              WinSendDlgItemMsg (hwnd, parity, BM_SETCHECK, 1, 0);
              WinSetFocus (HWND_DESKTOP, WinWindowFromID
                                                   (hwnd, parity));
              RETURN 1;
        |  WM_CONTROL:
              parity := LOWORD (mp1);
              RETURN 0;
        |  WM_COMMAND:
              WinDismissDlg (hwnd, 1);
              RETURN 0;
        ELSE
           RETURN WinDefDlgProc (hwnd, msg, mp1, mp2);
        END;
     END;
  END ParityDlgProc;
```

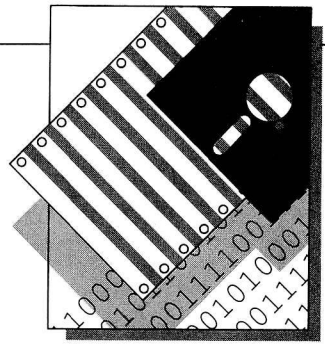
(continued on page 116)

(Listing continued)

```
967:      PROCEDURE AboutDlgProc ['AboutDlgProc'] (
968:      (* Process "About" Dialog Box *)
969:            hwnd  : HWND;
970:            msg   : USHORT;
971:            mp1   : MPARAM;
972:            mp2   : MPARAM) : MRESULT [LONG, LOADDS];
973:         BEGIN
974:            IF msg = WM_COMMAND THEN
975:               WinDismissDlg (hwnd, 1);
976:               RETURN 0;
977:            ELSE
978:               RETURN WinDefDlgProc (hwnd, msg, mp1, mp2);
979:            END;
980:         END AboutDlgProc;
981:
982:
983:      PROCEDURE SendFNDlgProc ['SendFNDlgProc'] (
984:      (* Process Dialog Box that obtains send filename from user *)
985:            hwnd  : HWND;
986:            msg   : USHORT;
987:            mp1   : MPARAM;
988:            mp2   : MPARAM) : MRESULT [LONG, LOADDS];
989:         BEGIN
990:            CASE msg OF
991:               WM_INITDLG:
992:                  WinSetFocus (HWND_DESKTOP, WinWindowFromID
                                                        (hwnd, ID_SENDFN));
993:                  RETURN 1;
994:            |  WM_COMMAND:
995:                  WinQueryDlgItemText (hwnd, ID_SENDFN, 20, ADR (sFname));
996:                  WinDismissDlg (hwnd, 1);
997:                  RETURN 0;
998:            ELSE
999:               RETURN WinDefDlgProc (hwnd, msg, mp1, mp2);
1000:           END;
1001:        END SendFNDlgProc;
1002:
1003:
1004:     PROCEDURE PathDlgProc ['PathDlgProc'] (
1005:     (* Process Dialog Box that obtains directory path from user *)
1006:           hwnd  : HWND;
1007:           msg   : USHORT;
1008:           mp1   : MPARAM;
1009:           mp2   : MPARAM) : MRESULT [LONG, LOADDS];
1010:        BEGIN
1011:           CASE msg OF
1012:              WM_INITDLG:
1013:                 WinSetFocus (HWND_DESKTOP, WinWindowFromID
                                                        (hwnd, ID_DIRPATH));
1014:                 RETURN 1;
1015:           |  WM_COMMAND:
1016:                 WinQueryDlgItemText (hwnd, ID_DIRPATH, 60, ADR (Path));
1017:                 WinDismissDlg (hwnd, 1);
1018:                 RETURN 0;
1019:           ELSE
1020:              RETURN WinDefDlgProc (hwnd, msg, mp1, mp2);
1021:           END;
1022:        END PathDlgProc;
1023:
1024:
1025:     PROCEDURE DirEndDlgProc ['DirEndDlgProc'] (
1026:     (* Process Dialog Box to allow user to cancel directory *)
1027:           hwnd  : HWND;
1028:           msg   : USHORT;
1029:           mp1   : MPARAM;
1030:           mp2   : MPARAM) : MRESULT [LONG, LOADDS];
1031:        BEGIN
1032:           IF msg = WM_COMMAND THEN
1033:              WinDismissDlg (hwnd, 1);
1034:              RETURN 0;
1035:           ELSE
1036:              RETURN WinDefDlgProc (hwnd, msg, mp1, mp2);
1037:           END;
1038:        END DirEndDlgProc;
1039:
1040:
1041:     PROCEDURE HelpDlgProc ['HelpDlgProc'] (
1042:     (* Process Dialog Boxes for the HELP *)
1043:           hwnd  : HWND;
1044:           msg   : USHORT;
1045:           mp1   : MPARAM;
1046:           mp2   : MPARAM) : MRESULT [LONG, LOADDS];
1047:        BEGIN
1048:           IF msg = WM_COMMAND THEN
1049:              WinDismissDlg (hwnd, 1);
1050:              RETURN 0;
1051:           ELSE
1052:              RETURN WinDefDlgProc (hwnd, msg, mp1, mp2);
1053:           END;
1054:        END HelpDlgProc;
1055:
1056:
1057:     PROCEDURE KeyTranslate (mp1, mp2 : MPARAM; VAR c1, c2 : CHAR)
                                                              : BOOLEAN;
1058:     (* Translates WM_CHAR message into ascii keystroke *)
1059:
1060:        VAR
1061:                                  code : CARDINAL;
1062:                                  fs : BITSET;
1063:                                  VK, KU, CH, CT : BOOLEAN;
1064:
1065:        BEGIN
1066:           fs := BITSET (LOWORD (mp1));                    (* flags *)
1067:           VK := (fs * BITSET (KC_VIRTUALKEY)) # {};
1068:           KU := (fs * BITSET (KC_KEYUP)) # {};
1069:           CH := (fs * BITSET (KC_CHAR)) # {};
1070:           CT := (fs * BITSET (KC_CTRL)) # {};
1071:           IF (NOT KU) THEN
1072:              code := LOWORD (mp2);   (* character code *)
1073:              c1 := CHR (code);
1074:              c2 := CHR (code DIV 256);
1075:              IF ORD (c1) = 0E0H THEN   (* function *)
1076:                 c1 := 0C;
1077:              END;
1078:              IF CT AND (NOT CH) AND (NOT VK) AND (code # 0) THEN
1079:                 c1 := CHR (CARDINAL ((BITSET (ORD (c1))
                                                  * BITSET (1FH))));
1080:              END;
1081:              RETURN TRUE;
1082:           ELSE
1083:              RETURN FALSE;
1084:           END;
1085:        END KeyTranslate;
1086:
1087:
1088:     PROCEDURE WindowProc ['WindowProc'] (
1089:     (* Main Window Procedure--Handles message from PM and elsewhere *)
1090:           hwnd  : HWND;
1091:           msg   : USHORT;
1092:           mp1   : MPARAM;
1093:           mp2   : MPARAM) : MRESULT [LONG, LOADDS];
1094:
1095:        VAR
1096:           ch : CHAR;
1097:           hps       : HPS;
1098:           pswp      : PSWP;
1099:           c1, c2    : CHAR;
1100:
1101:        BEGIN
1102:           CASE msg OF
1103:              WM_HELP:
1104:                 IF TermMode THEN
1105:                    WinDlgBox (HWND_DESKTOP, hwnd, HelpDlgProc,
1106:                               0, IDM_TERMHELP, 0);
1107:                 ELSE
1108:                    WinDlgBox (HWND_DESKTOP, hwnd, HelpDlgProc,
1109:                               0, IDM_HELPMENU, 0);
1110:                 END;
1111:                 RETURN 0;
1112:           |  WM_SETFULL:
1113:                 SetFull;
1114:                 RETURN 0;
1115:           |  WM_SETRESTORE:
1116:                 SetRestore;
1117:                 RETURN 0;
1118:           |  WM_SETMAX:
1119:                 SetMax;
1120:                 RETURN 0;
1121:           |  WM_MINMAXFRAME:
1122:                 pswp := PSWP (mp1);
1123:                 IF BITSET (pswp^.fs) * BITSET (SWP_MINIMIZE) # {} THEN
1124:                    (* Don't Display Port Settings While Minimized *)
1125:                    WinSetWindowText (FrameWindow, ADR (Title));
1126:                 ELSE
1127:                    WinSetWindowText (FrameWindow, ADR (Banner));
1128:                    IF TermMode AND
1129:                     (BITSET (pswp^.fs) * BITSET (SWP_RESTORE)#{}) THEN
1130:                       (* Force window to be maximized in terminal mode *)
1131:                       WinPostMsg (FrameWindow, WM_SETMAX, 0, 0);
1132:                    ELSIF (NOT TermMode) AND
1133:                     (BITSET (pswp^.fs) * BITSET (SWP_MAXIMIZE)#{}) THEN
1134:                       (* Prevent maximized window EXCEPT in terminal mode *)
1135:                       WinPostMsg (FrameWindow, WM_SETRESTORE, 0, 0);
1136:                    ELSE
1137:                       (* Do Nothing *)
1138:                    END;
1139:                 END;
1140:                 RETURN WinDefWindowProc (hwnd, msg, mp1, mp2);
1141:           |  WM_CREATE:
1142:                 hdc := WinOpenWindowDC (hwnd);
1143:                 VioCreatePS (frame_hvps, 25, 80, 0, FORMAT_CGA, 0);
1144:                 VioCreatePS (child_hvps, 16, 40, 0, FORMAT_CGA, 0);
1145:                 DosCreateThread (TermThrProc, TermThr, ADR
                                                  (TermStack[STKSIZE]));
1146:                 DosSuspendThread (TermThr);
1147:                 RETURN 0;
1148:           |  WM_INITMENU:
1149:                 Check (ColorSet);
1150:
1151:                 RETURN 0;
1152:           |  WM_COMMAND:
1153:                 DoMenu (hwnd, mp1);
1154:                 RETURN 0;
1155:           |  WM_TERMQUIT:
1156:                 TermMode := FALSE;
1157:                 DosSuspendThread (TermThr);
1158:                 VioAssociate (0, hvps);
1159:                 (* Restore The Window *)
1160:                 SetRestore;
1161:                 Enable (IDM_CONNECT);
1162:                 Enable (IDM_SEND);
1163:                 Enable (IDM_REC);
1164:                 Enable (IDM_DIR);
1165:                 Enable (IDM_OPTIONS);
1166:                 Enable (IDM_COLORS);
1167:                 RETURN 0;
1168:           |  WM_TERM:
1169:                 PutPortChar (CHR (LOWORD (mp1)));   (* To Screen *)
1170:                 RETURN 0;
1171:           |  WM_CHAR:
1172:                 IF TermMode THEN
1173:                    IF KeyTranslate (mp1, mp2, c1, c2) THEN
1174:                       PutKbdChar (c1, c2);   (* To Port *)
1175:                       RETURN 0;
1176:                    ELSE
1177:                       RETURN WinDefAVioWindowProc (hwnd,msg,mp1,mp2);
1178:                    END;
1179:                 ELSE
1180:                    RETURN WinDefWindowProc (hwnd, msg, mp1, mp2);
1181:                 END;
1182:           |  WM_PAINT:
```

```
                hps := WinBeginPaint (hwnd, NULL, ADDRESS (NULL));
                GpiErase (hps);
                VioShowPS (25, 80, 0, hvps);
                WinEndPaint (hps);
                RETURN 0;
          |  WM_SIZE:
                IF TermMode THEN
                   RETURN WinDefAVioWindowProc (hwnd, msg, mp1, mp2);
                ELSE
                   RETURN WinDefWindowProc (hwnd, msg, mp1, mp2);
                END;
          |  WM_DESTROY:
                VioDestroyPS (frame_hvps);
                VioDestroyPS (child_hvps);
                RETURN 0;
          ELSE
             RETURN WinDefWindowProc (hwnd, msg, mp1, mp2);
          END;
       END WindowProc;


    PROCEDURE ChildWindowProc ['ChildWindowProc'] (
    (* Window Procedure for Send/Receive child windows *)
       hwnd : HWND;
       msg  : USHORT;
       mp1  : MPARAM;
       mp2  : MPARAM) : MRESULT [LONG, LOADDS];

       VAR
          mp : USHORT;
          hps : HPS;
          c1, c2 : CHAR;

       BEGIN
          CASE msg OF
             WM_PAINT:
                hps := WinBeginPaint (hwnd, NULL, ADDRESS (NULL));
                GpiErase (hps);
                VioShowPS (16, 40, 0, hvps);
                WinEndPaint (hps);
                RETURN 0;
          |  WM_CHAR:
                IF KeyTranslate (mp1, mp2, c1, c2) AND (c1 = ESC) THEN
                   Aborted := TRUE;
                   RETURN 0;
                ELSE
                   RETURN WinDefWindowProc (hwnd, msg, mp1, mp2);
                END;
          |  WM_PAD:
                mp := LOWORD (mp1);
                IF (mp = PAD_Error) OR (mp = PAD_Quit) THEN
                   WriteL.,
                   IF mp = PAD_Error THEN
                      WinMessageBox (HWND DESKTOP, hwnd,
                                     ADR ("File Transfer Aborted"),
                                        ADR (Class), 0, MB_OK
                                                      + MB_ICONEXCLAMATION);
                ELSE
                   WinMessageBox (HWND_DESKTOP, hwnd,
                                  ADR ("File Transfer Completed"),
                                  ADR (Class), 0, MB_OK
                                                + MB_ICONASTERISK);
                END;
                DosSleep (2000);
                VioAssociate (0, hvps);
                WinDestroyWindow(ChildFrameWindow);
                Enable (IDM_CONNECT);
                Enable (IDM_SEND);
                Enable (IDM_REC);
                Enable (IDM_DIR);
                Enable (IDM_OPTIONS);
                Enable (IDM_COLORS);
             ELSE
                DoPADMsg (mp1, mp2);
             END;
             RETURN 0;
          |  WM_DL:
                DoDLMsg (mp1, mp2);
                RETURN 0;
          |  WM_SIZE:
                WinSetWindowPos (ChildFrameWindow, 0,
                   Pos.cx DIV 4, Pos.cy DIV 4,
                   Pos.cx DIV 2, Pos.cy DIV 2 - 3,
                   SWP_MOVE + SWP_SIZE);
                RETURN WinDefAVioWindowProc (hwnd, msg, mp1, mp2);
          ELSE
             RETURN WinDefWindowProc (hwnd, msg, mp1, mp2);
          END;
       END ChildWindowProc;


BEGIN   (* Module Initialization *)
    WITH Settings[ID_COM1 - COM_OFF] DO
       baudrate := ID_B1200;
       parity := ID_EVEN;
       databits := ID_DATA7;
       stopbits := ID_STOP1;
    END;

    WITH Settings[ID_COM2 - COM_OFF] DO
       baudrate := ID_B19K2;
       parity := ID_EVEN;
       databits := ID_DATA7;
       stopbits := ID_STOP1;
    END;
    PrevComPort := NONE;
    comport := ID_COM1;
    TermMode := FALSE;(* Not Initially in Terminal Emulation Mode *)
END Shell.
```

End Listings



Do You Have a Future in Vegetative Virtuality?

Michael Swaine

My garden will never make me famous,
I'm a horticultural ignoramus,
I can't tell a string bean from
a soybean,
Or even a girl bean from a boy bean.
— Ogden Nash

I think that I see an undiscovered product niche. It's a vertical product niche, with thousands of potential customers and the prospect of multiple sales to each customer. The niche is virtual shrubs.

Macintosh-Aided Design magazine recently ran a good article on landscape design, 3-D graphics, and vegetable verisimilitude. Daniel Earle's "Three-Dimensional Representation of Plants in Landscape Design" (June, 1990) described how landscape designers and architects can use 3-D modeling tools to produce the perspective views of trees and shrubs necessary in their work.

Current 2-D object-based approaches, according to Earle, allow creating libraries of plant symbols, and freehand drawing programs produce acceptable freehand results; but 3-D modeling approaches have not been impressive: *Present computer approaches that create plants that look like lollipops, rockets, or shaped concrete do not adequately satisfy planting design considerations.* It's necessary that plant forms exhibit freedom, variety, and organic form, otherwise they look like extensions to the *hardscape* around them, rather than offering a contrast.

Earle proposes using 3-D graphic tools to model trees and shrubs, rather than just patching together a few geometric forms. He identifies essential elements of structure and form of plants, and shows how to develop, without excessive effort, *workable plant forms.*

It struck me as I read the article that it ought to be possible to do better than workable plant forms and that it ought to be easier. Earle's approach definitely leads to better-looking trees and shrubs, but it's ultimately not different in kind, but only in degree, from the 2-D plane geometry models he criticizes. Trees aren't really made up of plane geometry objects and conic sections. Why not use fractals?

As I've pointed out here before, fractal geometry, the new branch of mathematics that Lucas films has used to produce pseudonatural forms in science fiction movies, is remarkably good at modeling natural forms such as coastlines, mountain ranges, and foliage. Certain classes of fractals are particularly good at mimicking trees and shrubs. This is because, so far as I can understand it, the recursive growth pattern of the fractal reflects a similar growth pattern in nature. Fractals know some of Nature's algorithms.

My own explorations through the maze of these shrubby fractals have run up against two thorny hedges: performance and predictability. Fractal algorithms are viciously recursive, and any more than a few levels of recursion takes unacceptably long to execute on my hardware (and more to the point, in my code, but improving the code will only shift the performance barrier back a few levels). Further, although I have generated some attractive and natural-looking plant forms, the process is hit-or-miss: I don't know the relationship between the parameters I can manipulate and the defining features of the sorts of plants I can model.

For the needs of landscape designers, it seems to me, the performance problem is not critical. Their needs are for simple but evocative and recognizable images. A picture should be recognizably a birch tree or a particular variety of succulent, but rich detail is not desirable. Even the simple algorithms and implementations I've played around with should be able to meet these performance requirements.

If the predictability problems were solved, fractals would, I think, take over the field. The advantages are compelling:

- Greater naturalness. The controlled randomness of fractal techniques gives a more natural image than does building from conventional geometric forms.
- Deep naturalness. Fractal geometry apparently accurately models the process of growth and the underlying grammar of the natural forms it simulates.
- Greater efficiency. Fractal images have absurdly modest storage requirements, if stored in the form of the generator function and generated at need.
- Greater power. Fractal techniques use recursive image generation to allow any level of detail to be produced from a single stored generator function.

Now, apparently, the predictability problem has been solved. Michael Barnsley, co-founder of Iterated Systems, Norcross, Georgia, has found that by starting with a digitized natural image, say, of a coastline, and mapping segments of it to a large library of stan-

dard fractal-like transformations, you can turn the image into a formula that does a credible job of reproducing the image, at an acceptable video rate, with a compression ratio of perhaps 500:1! Barnsley has turned the insight into a business, but he doesn't seem to be filling the niche I think I see, and the basic insight is not proprietary.

So I think that somebody is going to make money selling libraries of virtual shrubbery. Libraries of natural-looking trees and shrubs ought to speed the landscape designer's work, and ought to sell well.

The simplest implementation of the idea would be to set up a plant plant (wait, make that a virtual shrubbery nursery), and to crank out libraries of plant forms in a standard 3-D format. This approach, though, would lose some of the benefits of using fractals. The desired amount of detail could vary, and a fractal generator function can lay on as much detail as you can sit still for. Perhaps a better distribution model would be as a *smart* picture; some sort of object, containing the generator function, an interface allowing the input of a recursion level, and the minimal code required to execute the generator function, to unfold the function into a graphic image. The recursion level would control the degree of detail in the resulting image.

By creating such *smart* plants, one would belie the old saying, "You can speed up horticulture but you cannot make it think."

Is Compiled Hype Faster than Interpreted Hype?

After three years we still don't know what HyperCard is, but it's now a full-blown product category. I don't mean

Certain classes of fractals are particularly good at mimicking trees and shrubs because the recursive growth pattern of the fractal reflects a similar growth pattern in nature

stackware; I mean — well, that category of uncategorizable tools that people are using to do product prototyping or to produce online help systems, front-ends to remote databases, stand-alone kiosk systems, as well as simple databases of text, pictures, sounds, and code. Apple's HyperCard, Silicon Beach's SuperCard, Symmetry's Plus, Asymmetrix's ToolBook: I've taken to calling these tools hypermedia authoring systems, a pompous-sounding expression that is, at least, nebulous enough to capture a lot of their uses.

And they have a lot of uses; or better, people are putting them to a lot of uses. These products, including ToolBook, which is only a few months old, have bent to winds from all quarters. ToolBook is already in use, for example, in hardware and software firms; Mac developers, traditional DOS vendors, and mainframe software vendors; multimedia software vendors, database companies, and compiler vendors.

The mere fact that people are using them for a wide variety of purposes doesn't mean that they are good tools for any of those purposes. In fact, an argument could be made that — within certain limits — widespread use of a development tool is evidence of its unsuitability for serious development work. There are, after all, only so many serious developers in the world, and

(continued from page 120)
any tool being used by more people than there are serious developers is ipso facto not a serious development tool. The argument is slippery, but the conclusion it reaches is one that most serious developers would accept: HyperCard and its ilk are not serious development tools.

I can't challenge the conclusion head on, but there are degrees of seriousness, and I'd like to present the case for an important increase in the seriousness of these tools. I have just finished writing reviews, for various publications, of the latest versions of all four of these products (the latest version was also the first version in the case of ToolBook). My cumulative impression after doing all the reviews is that the current versions of these products represent a new level of seriousness for the product category; that's my impression, and I'll present the evidence so you can decide.

Apple took a long time in releasing HyperCard 2.0, but managed to justify the wait. If you are familiar with HyperCard 1.x but haven't seen 2.0, you should appreciate the new features. If you are unfamiliar with HyperCard, you may be surprised at the essential elements of a development tool that it was missing. Either way, it should be clear that the new version is a significant improvement.

First, HyperCard now gives much greater control over the user interface. Formerly, it restricted you to the HyperGhetto of a single window of fixed size, menu structure, and appearance. You could always spot a HyperCard application. Now, you can create arbitrary-sized and -shaped windows, open several windows (hence several stacks) simultaneously, and control the menus completely. You can create external windows (more about that momentarily) and floating palettes. You can also trap all keyboard events.

Second, the development environment is more serious. HyperTalk is now compiled, the script editor is more useful, and there's an integrated debugger.

The modal dialog box that served as a script editor in HyperCard 1.x was one of its more annoying features. HyperCard 2.0 has a new script editor that fully supports the edit menu, horizontal scrolling, multiple script editing windows, search and replace, and triple-clicking to select a line. You can comment or uncomment blocks of code automatically. The script editor uses document windows, so you can move freely between the card window and the script editor window. HyperTalk code is now compiled to RAM rather than being interpreted, and only source code is saved on disk; the script editor always loads the script from disk to edit it.

The script editor is integrated with a debugger, and errors now bring up a dialog box allowing you to jump to the offending line in the script under the script editor or under the debugger. The debugger supports setting multiple permanent or temporary breakpoints, and there are two windows for tracking the values of variables and all messages sent. You have some options in what messages get tracked: Unused and idle messages are by default not tracked, but can be.

Third, there's a new external command (XCMD) interface, which Apple used to create the new development environment. The script editor, debugger, message watcher, and variable watcher all are XCMDs. As a result, you can replace any of them with third-party tools simply by changing the name of a global variable. The new XCMD interface is different enough that existing XCMDs that make assumptions about memory allocation may not work with HyperCard 2.0. But the new interface is such an improvement that most existing XCMDs probably ought to be rewritten to take advantage of its new features, anyway.

XCMDs can now run concurrently with each other and with HyperCard. You can set an XCMD's priority by specifying the time between idle-time calls to the XCMD. Note, too, that HyperTalk scripts now run in the background under MultiFinder.

XCMDs can now create and manage their own windows. These new external windows can be document windows or windoids, and can use color if color QuickDraw is available. Any event pertaining to the window, such as *mouseDown* or *activateEvt*, will be passed to the XCMD that created it. The standard HyperTalk get and set syntax for properties has been extended to external windows. Mostly, the XCMD controlling the window should handle the properties of the window, but HyperTalk will handle the *loc* and visible properties of an external window if its XCMD doesn't. A separate tool, freely available but not packaged with HyperCard 2.0, will support the creation of palette windows.

Fourth, the language has been enhanced in many ways that increase programming flexibility and power. The message hierarchy can be modified. The *do* command now handles multiline containers. The commands *start using* and *stop using* manage the addition and deletion of stacks to the message-passing hierarchy; up to ten stacks can be added between the present stack and the Home stack. And there is now syntax in HyperTalk for checking for the existence of an object, similar to SuperTalk's *exists* function.

Commands that in the past brought up a modal dialog box and therefore could not be automated have been changed to allow dialog box choices to be passed as parameters. Dialog boxes now size themselves to the text supplied (up to approximately 240 characters). In the case of the *ask* command, this applies to the prompt string and the default response. And all dialog boxes can be dismissed by Command-period.

ToolBook has capabilities that make it particularly interesting, partly the result of its knowing how to use DOS and Windows capabilities. While I've enjoyed playing with the toy applications Apple supplied to developers to demonstrate the interapplication communications capabilities coming someday in System 7, I've actually written ToolBook applications that message one another via DDE, doing today what Apple is promising for next year. And ToolBook's DLLs are a step beyond even HyperCard's new XCMD interface.

Like so many other things, this category of software has been made more

(continued from page 123)
significant by the release of Windows 3.0. HyperCard stacks will soon be portable to ToolBook format, ToolBook runtime is being bundled with Windows, Zenith is bundling ToolBook with all its 386 machines. If there was a market for the products you develop with these tools, and there was (it just wasn't the market a lot of people thought it would be), then the market is suddenly an order of magnitude larger. I'm taking it seriously enough; as I've mentioned here before, I've been putting together a newsletter for users of hypermedia authoring tools, and the first issue is finally about to go to press. Working on the newsletter, talking to the developers, I began to see that something is indeed afoot. ToolBook was released at a press conference featuring 26 third-party developers and has already spawned a dedicated newsletter; and HyperCard 2.0 comes out the chute with three books, and more to come (my HyperCard bookshelf has over two dozen volumes).

Of course, given the memory demands of these tools, one could argue that it's Macintosh, System 7 and Microsoft Windows that are giving them credibility, by forcing everyone to install more memory.

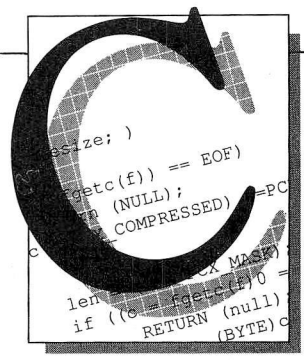
Further Reading

For more information on fractal horticulture, see Daniel Earle, "Three-Dimensional Representation of Plants in Landscape Design," *Macintosh-Aided Design*, June, 1990; A.K. Dewdney, "Mathematical Recreations: Creating Fractal Landscapes With a Home Computer," *Scientific American*, May, 1990; or Mike Swaine, "Fractals: A New Dimension in Scripting," *HyperLink* magazine, November-December, 1989. Or send for Barnsley's book, *Fractals Everywhere*, or his book-and-disk system, *The Desktop Fractal System*, both from Academic Press.

For more support for my argument on the maturation of hypermedia authoring tools, see my article, "Author! Author!" *Personal Workstation*, August, 1990; my reviews of the latest versions of SuperCard, Plus, and HyperCard, slated for the October issue of *MacUser* magazine; or issue #1 of my new hypermedia authoring journal, *HyperPub*. For details, write to "The Prose Lab," Dept. DDJ, 31 Patrick Road, Santa Cruz, CA 95060. Or call me at 408-459-8564.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 10.**



Hacks, Spooks, and Data Encryption

Al Stevens

The lexicon keeps mutating. In days gone by "hackers" were honored souls who rummaged in the innards of computer systems finding clever and tricky ways to do things within the bounds of official sanction. A good algorithm was called a "good hack." A generation of computer invaders and the media have likewise pilfered the word so that now a hacker is someone who steals computer time and information. This column is about how the hackers of yore can protect their users from the new generation of punk hackers. We'll discuss the basics of data encryption and implement two encryption algorithms in C.

I just finished reading *The Cuckoo's Egg* by Clifford Stoll (1989, Doubleday). It's not about C (although C is mentioned and Unix is a major player), but all programmers will enjoy and relate to the true story about a mysterious computer break-in artist and the programmer who relentlessly sniffed out the devious interloper's location and identity. It's a compelling book. Plan enough time to read it in one sitting because you'll need it. Stoll is the programmer, and he writes about how a seemingly mundane assignment to find a 75 cent discrepancy in his employer's computer accounting system turned into a one year trek through the holes in Unix and VMS, the careless neglect and ignorance that most system managers have about system security, and the mindless, motionless bureaucracies of federal law enforcement and intelligence agencies.

You will learn from this book about computer pirates and how they routinely gain unauthorized access to computer systems by finding the unlocked back doors and unplugged holes. You'll also learn how the computers of government, universities, and research centers are linked together in complex networks by Tymnet and other communications services. And, if you've never worked inside your government, you'll gain from Stoll's novitiate insight into the bewildering and confounding nature of Potomac fever and lethargy. Character development is weak, and some of the dialogue is obviously contrived to explain things, but the book is loaded with suspense, and you will not want to put it down.

Data Encryption

Reading *The Cuckoo's Egg* brings us to the subject of data encryption. Data encryption means that you take some entity of data that has meaning in its original form and mangle it beyond recognition so that unauthorized peeping persons can neither understand nor use it. Later, you or someone who has authorized access to the information can unscramble it and use it. The idea, of course, is to scramble it in a way known only to those of the inner sanctum. The stuff of spies.

Computer-aided encryption involves at the very least an algorithm and often much more. The complexity of your encryption scheme will depend on the measure of protection you require or your level of paranoia, whichever is greater. For a well-developed treatment of the various techniques of encryption, see "Cloak and Data" by Rick Grehan in *Byte Magazine*, June 1990.

The other side of encryption is decryption. Usually, but not always, you need to decrypt something in order to use it. When would you not? The sign-on password is an example. When you select a password for access to a system, the computer will often encrypt it before it records it, forgetting the original. The encryption algorithm is designed so that no one can derive the original password from its encrypted form. If someone gets hold of the password file, they cannot use its contents to sign onto someone else's account. When you sign on, the system encrypts the password you enter and compares it to what is stored. This approach prevents the system from making a permanent record of passwords.

Stoll compared such password encryption schemes to sausage grinders. You can't run the ground meat backwards through the grinder and get a sausage. And so he wondered why his ersatz user kept downloading the useless password files. Later he learned that the Unix-encryption algorithm is common knowledge, and his anonymous burglar had used the algorithm to encrypt all the words in an English spelling dictionary. He would compare the downloaded encrypted passwords with the ones from his dictionary and thus learn the passwords of those users who innocently selected English words.

There are some very practical uses for data encryption in the world of shared computers. Every complex system must have a superuser, a system manager, a supervisor account, or whatever, so that the systems programmers can maintain the operating system. Some other users — users of applications — are responsible for information so sensitive that it must not be seen by others, not even the system supervisor. Payroll records, cost proposals, hostile takeover bids, employee health records, and such are all compartmentalized information to be viewed only by those who have the need to know.

Many successful unauthorized system penetrations involve the perpetrator's ability to gain superuser status through knowledge of a bug in the system. With that status, all the files

and programs are wide open to the spook. Encryption adds one more level of protection against inquiring minds.

Electronic mail is a hot new item among local and wide area networks. With all that text roaming around all those systems with so many varied degrees of protection and privilege, there is always a potential for that confidential memo, love letter, or football pool to be seen by the wrong eyes. Trouble usually follows.

The complexity of your encryption scheme will depend on the measure of protection you require or your level of paranoia, whichever is greater

As an example, the Novell network provides a measure of protection by allowing the system supervisor to grant read and write privileges to users at the network subdirectory level. If the system is properly configured, I can write to your mail subdirectory but I cannot read it. I can read the subdirectory where the software is stored but I cannot write it. I can read and write to my own mail subdirectory. When mail leaves my local area network to go to someone somewhere else, I must depend on the integrity of that remote site to similarly protect my precious data from prying eyes. Let's make the absurd assumption that all systems are secure and that all system supervisors are trustworthy. Is there a back door into this scheme of protection, one that would admit an intruder? Of course there is.

Novell network applications exchange messages between networks by using a delivery agent program called the Message Handling Service (MHS). MHS has become a standard, and because of its acceptance, other platforms such as MCI Mail and CompuServe have or are developing gateway processes that exchange messages between themselves and MHS installations. Big stuff. Now suppose that you and I are common, unprivileged users on a network that uses MHS and I want to intercept your incoming mail. Our network has a hub name and password that the MHS administrator assigns. The hubs call one another and use those data elements to identify and verify one another. All I need to do to get your mail is set up another system, assign it our hub name and password and have it connect to the guys who send stuff to you. How do I learn the hub's password? Simple. MHS records it in the public directory file that everyone can read. Dumb. But typical. And reason enough to consider adding encryption to our message exchanging systems.

One- and Two-Key Encryption

Usually you encrypt a file by specifying a key value to the encrypting algorithm. The intended receiver of the file must know the key value in order to decrypt and use the data. This often presents a dilemma. If you want to remember only one key — mind, you mustn't write it down or keep it in a file — but have a lot of people to send mail to, then every one of them must know your key. The more of them there are, the greater potential there is for compromise and the less substantial the protection. If you exchange mail with a lot of people who do not know your key, then you need to remember all their keys. Although propriety prevents you from making any kind of permanent record, no one can remember so many keys, and the stuff gets written down somewhere.

A better scheme for encryption in electronic mail systems involves two keys, a public one and a private one. Your public key is known to everyone and only you know the private one. On the other hand you know the public keys of the other users. The public key is a function of the private key, and like the password discussed above, one cannot usually reverse-engineer the private key from the public one. Everyone sends things to you that are encrypted with the public key. But only the private key can decrypt the messages. You need to remember only one key for yourself, and you can record the public keys of your correspondents wherever you like, even in your electronic mail address book. Grehan's article in *Byte* explains several algorithms for developing a two-key encryption system.

In one-on-one exchanges, a single key system seems to work better. You and your pen pal know the key, and no one else does. You freely exchange encrypted messages. When it is necessary, the two of you agree to change the key.

CRYPTO

Listing One, page 147, is crypto.c, an implementation of a small encryption algorithm. It's all the encryption most

(continued from page 128)
of us would ever need. Unless you are in a really hostile environment where highly sensitive records are kept, encryption usually serves only to keep the curious honest. In Stoll's book, although the intruder tried for a year to get at something classified, he never did. The lax security he consistently found allowed him to purloin only from unclassified systems.

CRYPTO reads a key word and two filenames from the command line. If you run the program against an encrypted file, CRYPTO decrypts it. The algorithm is simple. The program reads the input file in 64-bit blocks, does an exclusive OR of the block and the keyword, and writes the block to the output file. To decrypt the encrypted file, you run it against the same CRYPTO program by using the same password.

To the uninitiated, the encrypted file seems to contain random byte values. Even if a snooper has the CRYPTO program, without the keyword the file is useless. This algorithm is much better than a simple character substitution encryption scheme because each character's modification is a function of its position in the 8-byte block and the corresponding character in the keyword. This technique is not immune to code-breaking techniques, but it will keep your code-breaker busy for a while. Encryption sometimes serves its purpose if it merely delays when the information goes public. If your message says, "We attack at high noon," who cares if the code-breakers figure it out by 1 P.M.?

Data Encryption Standard

The government has sanctioned a standard algorithm for single-key encryption called the Data Encryption Standard (DES). You can get a copy of the standard from most government libraries by requesting the Federal Information Processing Standards Publication (FIPS PUB) 46. The algorithm is complex, and FIPS PUB 46 is not the clearest documentation you'll ever read. I'll try to explain it better and at the same time demonstrate it with C programs that implement the algorithm.

Encryption with DES

DES encrypts data 64 bits at a time. It mangles the 64 bits by way of a several-step algorithm that includes a 64-bit key value supplied by the user. The standard specifies a key of eight 8-bit bytes and does not use the most significant bit of each byte in the key, reserving that as a parity bit if needed.

DES works with bit strings and refers to bit 1, bit 2, and so on. You have to read the standard carefully all the way through to determine that bit 1 is the most significant bit of an 8-bit byte or of a 64-bit block and that bit numbers read from left to right. They are unspecific about the integer and long integer configuration of the architecture, so you must be careful about the last-byte-first view of data in most 16- and 32-bit machines.

The encryption algorithm works on the data file in 64-bit blocks. It begins by rearranging the bits in the block according to an initial permutation table. Then it separates the block into two 32-bit halves, a left half and a right half. There are 16 iterations of a mangling process designed to mutate the data beyond recognition with the key playing a major part. The result of these iterations is a final pair of 32-bit halves. The algorithm permutes this 64-bit block by using a permutation table that is the inverse of the initial permutation table. The output from this permutation is the encrypted 64-bit block. An interesting property of this inverse permutation is that if your input file does not use the most significant eighth bit, neither will the encrypted file. This means that if you transmit ASCII text data files over

(continued from page 130)
serial lines with 7-bit data byte communications protocols, your encrypted files will transmit correctly.

That seems simple enough. Now things begin to get complicated. The 16 iteration data-mangling process goes like this:

For each iteration, the left half of the block is exclusive ORed with the 32-bit output of a function called *f*. Then, except for the sixteenth iteration, the halves are exchanged. That's not so bad. Now let's consider the *f* function, which is appropriately named.

The *f* function takes the 32-bit right half of the block and the 48-bit output of the *KS* (key schedule) function as its arguments. The right half is called "R." The function permutes the 32 bits of R into 48 bits. This permutation is called "E," and it is exclusive ORed with the 48-bit output from the *KS* function. The 48-bit result is then segmented into eight 6-bit values. The *S* function compresses each of these a 4-bit value. The eight 4-bit values combine into one 32-bit value, which is then permuted through a permutation table called "P." That permuted value is the output from the *f* function. Whew.

The *S* function consists of eight different subfunctions (*S1, S2, S3* . . .) depending on whether it is compressing the first, second, third, etc. 6-bit block. There are eight tables, one for each subfunction. The tables are arrays of two dimensions with 4 columns and 16 rows. Each entry in the array contains a value between 0 and 15 representing the 4-bit compressed representation of the 6-bit input. The 6-bit value to be compressed provides the array subscripts in this bizarre manner: Bits 1 and 6 combine to be the row subscript 0 – 3. Bits 2 – 5 are the column subscript 0 – 15. The *S* function returns the 4-bit value taken from the proper array as the result of these subscripts. We're not done yet.

The government has sanctioned a standard algorithm for single-key encryption called the Data Encryption Standard (DES)

The *KS* function returns a 48-bit value based on the 64-bit key. Its arguments are the iteration number of the 16 iterations discussed earlier and the key value for the encryption. For the first iteration, the function permutes the key value by using the Permuted Choice 1 table. The key divides into two halves and these halves are each shifted one or two bits to the left depending on which iteration is running. A table controls the shift values. Each iteration after the first uses the shifted value of the previous iteration as its input, does its own shift, and then permutes the value once more by using the Permuted Choice 2 table.

Decryption with DES

Decryption reverses the encryption procedure. Because the initial and inverse-initials permutations are reciprocal, the decryption steps are the same except that the decryption reverses the half exchanges during the iterations and uses the permuted key values returned by the *KS* function in the reverse order of that used by encryption.

ENCRYPT and DECRYPT

Listing Two, page 147, is des.h, the header file that defines the global formats, function prototypes, and external data names. The file also defines the preprocessor macros that the pro-

gram uses to build permutation tables. See the discussion on permutations later for an explanation of these macros.

Listing Three, page 147, is encrypt.c, the program that scrambles a data file. You run it by entering the keyword, which should be eight characters, the name of the input file and the name of the encrypted output file. The program reads the file 64 bits at a time and encrypts those 64 bits with the DES algorithm. It permutes the input with the initial permutation table and does the 16 iterations of right- and left-half exchanges and key-dependent block modifications. Then it uses the inverse initial permutation table to permute the output.

Note that the program computes the 16-key schedules into an array at the beginning of the program. These values are constant for the entire process and do not need to be recomputed for every iteration that uses them.

The output file will always be an even multiple of 8 bytes. This is necessary because the encryption process scrambles in 64-bit chunks.

Listing Four, page 148, is decrypt.c, which is similar to encrypt.c except that it reverses the order of right-left exchanges and fetches from the key schedule array. The decrypted file is the length of the encrypted file, an even multiple of 8 bytes with the excess bytes padded with zeros. This padding could be a problem, particularly with files of fixed format where an application appends records. If this is the case, you could modify the encryption program to write a control value as part of the encrypted file. The value would specify the length of the original data file. You can modify the decryption program to use this value to truncate the excess bytes in the decrypted file. You could decide to bypass encrypting the last block if it is less than 8 bytes. Beware, though. If the user's message to his boss's secretary ends with "Love, Ed", there might be some explaining due.

Listing Five, page 148, is des.c, which contains the *DES* functions. I explained most of them in the discussion on the algorithms. The terse function names *f*, *S*, and *KS* come from the DES specification. The discussion on permutations that follows explains the permute and *inverse_permute* functions. The *sixbits* function might need some explanation. Its purpose is to extract and return a 6-bit value from within a 48-bit block. None of this is readily supported by the typical 8- and 16-bit architectures most of us use, so the *sixbits* function uses a brute force method. A table named "ex6" contains an entry for each of the eight 6-bit values. Each entry identifies the 2 bytes that contain the spread of 6 bits. One element specifies the number of bits to shift the byte left and another specifies the number to shift right. Those two elements are mutually exclusive. Another element specifies the mask for ANDing off the excess bits.

Listing Six, page 149, is tables.c. I put the tables into a separate source file because they compile so slowly. The next discussion explains why.

DES Permutations

I have mentioned the various permutations that DES performs. A permutation of a data block consists of rearranging the bits into a specified order. The DES specification provides tables that identify what the permuted order of the bits will be for each permutation. There are several ways that you could implement such permutations in C. I built an array of bit masks for each of the 64 bits in a permutation. Each array entry contains a 64-bit value with one bit set on. Its position in the table associates it with the bit that it will change. So, if the third table entry has its fifth bit set, then the permuted output will have its fifth bit set only if the input has its third bit set. For testing the 64 input bit positions, I built a general-purpose table with the bits set in ascending order — the first bit is set in the first entry, the second in the second, and so on. Some of this could be done more efficiently in assembly language with shifts and carry tests. The C language as implemented on the PC, however, does not have a convenient 64-bit shift operator. Consequently I chose the bit test mask and bit set mask approach described here.

To make building the bit masks easier and reduce the potential for transcription errors, I developed the preprocessor macros in DES.H. By coding the *p(n)* macro with a number between 1 and 64 as its argument, you tell the preprocessor to define a comma-separated set of eight integral values with seven of them of value 0 and one of them having a value with a single bit (1, 2, 4, 8, 0x10, 0x20, 0x40, or 0x80) set. The value 1 sets the left-most bit of the left-most byte. Subsequent ascending values set the next sequential bit to the right. The *p(n)* expressions can then be coded into the initializers for a character array.

By using this technique, I was able to lift the permutation table values directly out of the DES standard document and code them into the *p(n)* expressions. Observe the macros. When you code a *p(n)* expression, it expands to eight calls to the *b(n,r)* macro, one for each byte in the 64-bit mask. The *r*

argument identifies which of the 8 bytes the *b(n,r)* macro expands. The macro uses that value to test the range of the *n* argument to see if its corresponding bit falls within the byte. If not, the expression expands to a zero value for the byte. If so, the macro calls the *ps(n)* macro passing an argument in the range 1 – 8 that is computed from the *n* and *r* arguments to the *b(n,r)* macro. The *ps(n)* macro expands the byte to a value with a single bit shifted into the correct position.

This exercise illustrates the power of the C preprocessor. By coding these macros I am able to express the permutation tables with more readable language and less margin for error. But there is a cost. There are several of those tables in the program, and all that preprocessor activity slows a 20-MHz 386 to a crawl when the tables.c file compiles. The first time it happened, I thought that Turbo C had hung up.

DES Performance

The DES algorithm is complex and slow. The ENCRYPT program published here takes one minute, ten seconds to encrypt the text of this column, about 25K bytes, and the DECRYPT program uses the same time to decrypt it. I made these measures on a 20-MHz 386 with Turbo C 2.0 for the compiler. By comparison, the simpler CRYPTO program takes about 3.5 seconds to perform the same task. The poor showing of the DES programs could reflect a bad implementation (who, me?) or a non-optimum compile. I haven't tried it on a 4.77-MHz PC. Pack a lunch and bring a toothbrush.

The algorithms in DES.C assume a 32-bit long integer and an 8-bit byte. I used the PC's long integer to implement the halves of the 64-bit key and data block because it was convenient and offered the best chance for efficiency. A more portable program would adapt itself to the correct integral data type for the compiler and might not work as well. Perhaps the permutation functions could be optimized. A profiler will show that those functions occupy most of the algorithm's time. Optimization would, though, be compiler-dependent. You can look at the 64 iterations of the permute function and decide to use subscripts instead of pointers, or use 64 discrete in-line bit tests and ORs to avoid the loop overhead. Code it one way or another, and the program's performance could improve or degrade depending on the C compiler you use.

Because DES is defined by the National Bureau of Standards, a truly conforming device or program must be validated, and these programs have not been tested and blessed. My copy of the standard does not identify the validation procedures. Perhaps a good test would be to see if a candidate implementation can successfully exchange encrypted files with an approved device. The specification implies, however, that an implementer has some leeway in the design of the various permutation tables. If this is so, then the encrypted files would not be compatible between different implementations.

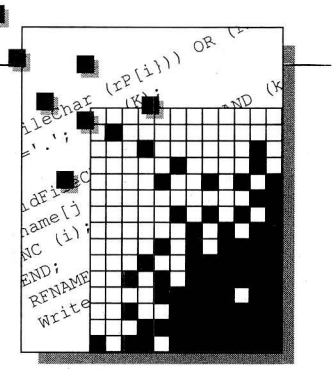
Epilogue

Farewell to Howard Benner, the author of TAPCIS. I never met Howard, but I use his work every day. So do many others. Howard was a programmer in a relatively new craft, one that is only about 40 years old. When I began programming 32 years ago, the programmers were all young, too, and there were not very many of us. I still think of us all as young people. It does not seem right somehow that programmers should die.

DDJ

(Listings begin on page 147.)

Vote for your favorite feature/article.
Circle Reader Service **No. 11.**



Pieces of Charlie

Jeff Duntemann KI6RA/7

While hunting for houses in Cave Creek, we saw a large-eared skinny brown dog trot across the highway close ahead of us with no hint of a look over its shoulder. He knew we were there; the ears said it all. This was obviously a creature that knew man and automobiles and wasn't much fazed by either. In short, we met the much-maligned coyote who, at the parting of the genes ages ago, chose the glorious life over the long one, perhaps because he somehow foresaw that we would take wolves and turn them into poodles.

For all his persecution, the coyote is doing surprisingly well in the Valley of the Sun. In defiance of conventional wisdom, there are (if my sources are correct), more coyotes living in North Phoenix today than there were 20 years ago, even with the incredible building growth the area has seen in that time. The two interlocking reasons aren't surprising when you think about them: More places to hide, and more things to eat.

Twenty years ago there were few culverts or storm drains in Phoenix. Now, after two hundred-year floods in close succession, culverts are everywhere, and they carry water perhaps three days out of the year. The rest of the time, they are bone-dry — perfect for waiting out the blazing Phoenix days or raising pups in perfect safety.

And things to eat, lordy . . . who'd chase a stringy old roadrunner when you can gorge on leftover French fries behind the dumpster at Carl's Junior? And for those atavist coyotes who prefer chasing down dinner, there is no shortage of free-roaming house cats. (Though fewer in recent years, I've heard: Nothing teaches a cat owner a sense of responsibility like finding pieces of Charlie underneath the palo verde tree in the empty lot down the street . . . .)

Ecological Niches

Actually, hearing of the Phoenix coyote population explosion surprised me less than hearing last year that Jensen & Partners was bringing out yet another C compiler, or this year hearing that JPI would be bringing out a new Pascal compiler. It seemed odd (or excessively gutsy) only at first. As with the coyotes' role in keeping cat doodoo out of my flower beds, I had overlooked an important ecological niche in the structured language world that was not being well-filled: The area of multi-language development.

The idea, in short, is to write a program in C and call routines written in Pascal . . . or to write a program in Pascal and call routines written in Modula2 . . . or some other permutation of the major languages. This is a difficult business for a great many reasons, and the major vendors haven't done much to make it easier. Microsoft started out down the right road years ago, by providing specific instructions on linking code written in their various languages, but they never took it very far, and even at its best the process was something I would consider an ordeal. Borland does provide some small .DOC files on calling Turbo Pascal from Turbo C and vice versa, but it seemed lots more trouble than it was worth.

Polymorphic Compilation

The TopSpeed solution to multi-language programming is bold: When you buy a TopSpeed language, you get the TopSpeed interactive development environment (IDE). When you buy a second (or third) TopSpeed language, it installs so that it can be invoked from within the same copy of the IDE (which is not installed a second time), as a peer with any other installed TopSpeed languages. The various languages recognize certain file extensions as their "own" and the IDE will automatically invoke the correct language when a given source file is specified for compilation. In other words, if you compile CHARLIE.C, the IDE invokes TopSpeed C without explicit instruction from you; or if you compile CHARLIE.MOD, the IDE will invoke TopSpeed Modula-2.

There is already a TopSpeed Assembler. TopSpeed Pascal is in its final testing stages, and may well be shipping by the time you read this. JPI has expressed intention of delivering a TopSpeed C++ and a TopSpeed Ada sometime in 1991, and both will fit right into the larger scheme.

Building CHARLIE.EXE from multiple pieces written in multiple languages requires a central blueprint, and this is provided by JPI's excellent automatic MAKE facility. Unlike MAKE utilities in the traditional C world, TopSpeed MAKE is built into the TopSpeed compilers. The programmer provides a project file containing instructions to the compiler: Memory model to be used, compiler options to invoke, which files are involved and must be linked together, and so on. The MAKE facility follows this plan to produce the final .EXE file, taking into account the time stamps of

involved source and object files so that it doesn't recompile anything needlessly.

Why Bother?

JPI has managed to get multi-language development over the line into the realm of the possible. But the question does remain, Why bother at all? Especially today, when you can do just about anything you want in just about any commercial implementation of any structured language. The reasons are few but they can be compelling:

1. Retaining existing investment in code libraries. Say you're in a shop that has always done its development in Modula2, and Corporate issues a proclamation that all future development is to be done in C. Hokay . . . except that that means that maybe five years' worth of existing libraries have to be rewritten. If you can rig things so as to call the existing Modula libraries from the new modules written in C, you can get the work done in C and convert the older libraries to C as time allows . . . if you bother at all.
2. Making use of specialty commercial libraries in different languages. Perhaps you do all your work in C, but the only third-party library you've found that provides a certain difficult function is in Pascal or Modula-2. A real-life example would involve the Solid Link Modula-2 library I described in my July column. I have numerous communications libraries for various languages in my collection, but Solid Link is the only one that implements the ZModem protocol, which is hairy in the extreme to implement yourself. By using TopSpeed C, you could have your C and ZModem too.
3. Maintaining multiple-platform libraries. If your shop must support different platforms than DOS, like the Sun under Unix, or the Amiga or Macintosh, it can make sense to identify what functions can be implemented identically across all platforms in a single highly-standard source file. User interface hassles might confine this to computational things such as fast Fourier transforms and so on, but it can still be worth doing. The only language all platforms might have in common is ANSI C, but a cross-language system would allow you to develop under DOS in the language of your choice and still use the common-platform C libraries.

On the flipside, there are things that are not reasons to work cross-language. This is the big one:

1. Working cross-language will not gain you execution speed. (Unless, of course,

one of the languages is assembler.) One of the significant happenings of the past few years is that code-generation technology has improved across the board, and there is no longer any automatic penalty for working in Modula-2 or Pascal. The recent Modula2 compilers from Stony Brook and JPI, in fact, are so good that you might

A memory model is a set of assumptions about how memory is addressed beneath the surface of a high-level language

incur a performance penalty for working in some implementations of C. Certainly, within the TopSpeed language family, you can assume that the code generation technology for all languages is similar and that code performance will be about the same no matter which language you're using.

And a lesser one:

2. Working cross-language will not give you additional functionality. If there's anything C can do that current commercial implementations of Modula-2 can't do, I've yet to see it. I'll provisionally say the same for JPI's own dialect of Pascal (which will not be a clone of Turbo Pascal) but we'll address this issue later on when I've had time to play with TopSpeed Pascal.

My own conclusion: If you need to work cross-language, JPI is currently the only way to go.

Memory Addressing and Memory Models

Working cross-language opens up a whole Pandora's box of things you have to keep straight to stay out of lockup-land. The single most important of these is the issue of memory models. Turbo Pascal people have rarely had to think about memory models, because Turbo Pascal only supports one memory model. (Much of the problem in converting old Turbo Pascal 3.0 apps to Turbo Pascal 4.0 and later lies in the fact that the conversion involves a change of memory model.) Most Modula-2 compilers have provided more than one memory model, but the majority of Modula-2 programmers choose the default memory model and simply stick with it to avoid having to understand what changing from one to another really means.

Simply put, a memory model is a set of assumptions about how memory is addressed beneath the surface of a high-level language. But before I go further down that road, let's review how memory is addressed in the 8086/8088 CPU and in real mode of the 286/386/486.

86-family real mode memory is limited to 2^{20} bytes, or 1,048,580 bytes, alias 1 Mbyte. An 86-family register, however, contains only 16 bits, which can specify only 216 (65,536 or 64K) locations when used as an address. How, then, do you address a full megabyte with 16-bit registers? The answer is to use two registers side-by-side. The high-order 16-bit register specifies one of 65,536 starting points within the megabyte of memory. Each starting point begins 16 bytes higher in memory than the one before it. The low-order register specifies an offset from some starting point. This offset may be up to 65,535 bytes away from the starting point.

The 65,536 bytes beginning at any starting point is called a "segment," and the number of the starting point that begins any given segment is called its "segment address." Segment 0 has its starting point at the very bottom of memory, in the very first byte of the memory system. Segment 1 begins 16 bytes up-memory from Segment 0. Segment 2 begins 32 bytes up-memory from Segment 0, and so on. Segment 65,535 begins only 16 bytes down from the very last byte in the 1,048,580 bytes addressable by the 86 family.

From this you should be able to see that by choosing the right segment and the right offset within that segment, you can uniquely specify any single byte in the whole 1,048,580 of them. Doing this choosing, however, requires two registers, which we generally call a "segment register" and an "offset register." The 86 architecture has several segment registers and several other registers that may act as offset registers. To pinpoint a location in a megabyte of memory, you have to put a segment address in a segment register, and an offset address in another register. Between the two of them, you can point anywhere in memory.

Looking Far and Near

Needless to say, the language compiler worries about all this so that you don't have to. The compiler takes care of keeping data variables in an area of memory that it knows how to find later on, and knows how to find a procedure or function in memory when that

procedure or function has to be called. The assumptions that a compiler uses to locate code and data comprise the memory model the compiler is currently using. There are several such models.

First of all, consider a program that has a lot of code but not much data.

If all the data a program will need to work with can fit into a single 64K segment, the compiler arranges things so that all data is put together within one segment, and makes the assumption that data will only be found in that segment. One segment register is given the segment address of this data segment when the program begins execution, and when data must be read or written, only the offset portion of the address must be changed. With only one register to modify, operations on data can be done more quickly than if both a segment and an offset address must be specified every time data is accessed.

When all of a program's data is placed together in a single 64K segment, we call it "near data."

Now, that same program has lots of code, more than will fit in a single segment. So the compiler sets up as many segments as it takes to contain all the program's code, and when one routine calls another, it must specify both the segment address and the offset address of the routine to be called. Code addressed this way is called "far code."

It can work the other way around for both code and data. If a program has only a little code, all that code can be placed together in a single segment with a single fixed segment address. All calls from one routine to another may then be made using a single 16-bit offset address. This scheme (called "near" code) uses less memory and is faster than when a full 32-bit address must be used. Similarly, programs with loads of data (say, several very large arrays) can arrange to place their data in multiple segments and address the data with full 32-bit addresses. This is somewhat slower and bulkier than near data, but it does allow you to use a great deal more data in a program.

All the Myriad Models

The memory model used by a compiler is predicated on what combination of code and data assumptions will be made. The memory model in which there is both near code and near data (and hence only two 64K segments) is called the "small model." The memory model in which there is both far code and far data is called the "large model."

The really ugly barrier to cross-language development, however, lies in something called "calling conventions"

In between are two intermediate stages called the "compact model" (near code, far data) and the "medium model" (far code, near data). See Table 1 for a summary of the various models.

There are two slightly peculiar mutant models, one on each end of the scale. If both code and data are small enough so that both can fit into the same 64K segment without tromping on one another, we call that the "tiny model." The tiny model's sole virtue is that it is the only memory model that can be massaged into a .COM file by the EXE2BIN DOS utility.

There is one more model, the "huge model," that's slightly tougher to explain. In all other models, there is an assumption that no single data item may be larger than a single segment. In other words, even though the large model may have as many data segments as it likes, no data item may span more than one segment. The huge memory model allows a single data item to span more than one segment.

This sounds simple, but when you start to mull what it means the whole concept starts to collapse. You can only

	Tiny	Small	Compact	Medium	Large	Huge
Code	Near	Near	Near	Far	Far	Far
Data	Near	Near	Far	Near	Far	Far
Max. prog. size	64K	128K	1MB	1MB	1MB	1MB

NOTE: The huge and large models differ primarily in how data is addressed; in the huge model, data items may span multiple segments.

Table 1: *The standard Intel 86-family memory models*

make sense of it by understanding how data is addressed by the CPU, and the best example is a large array.

An ordinary array in any data model but the huge model begins at some offset from the segment address. To read an item in the array, the CPU places the offset address of the array in an offset register, and calculates yet another offset based on the desired array index. For example, if the array is an array of records where each record is 32 bytes long, and you want to access the thirteenth element in the array, the CPU multiplies 32 by 12 (you count elements from 0) to calculate this second offset. The second offset is then added to the array offset to find the specific desired array element.

The problem should begin to come clear: The sum of the two offsets must still fit into a single 16-bit register to act as an offset from the array's segment address. A 16-bit register can only count to 65,535 — hence the array is limited to 64K in size.

In the huge memory model, the CPU must perform some considerably more sophisticated calculations to access any element of a "huge" array. Each element of the array has its own segment and offset address, and both must be calculated each time an element of the array is specified. Needless to say, this takes lots more time than when an array must fit into 64K.

Marrying Models

You have to keep all this stuff in mind when you begin to butt one piece of code from one language up against another piece of code from another language. If you call a piece of near code from a piece of far code, you'll probably crash the system. The near code pushes only one address (the offset address) onto the stack when the call is made, but the far code pops two addresses from the stack when it returns. It'll take the one address the calling code pushed, and grab the next two bytes on the stack as well, no matter what those 2 bytes actually are . . . and then launch off to the 32-bit address represented by the genuine offset address and the bogus segment address. Where it stops, well, nobody knows.

This might sound a touch familiar to Turbo Pascal people. In Turbo Pascal, calls made within a unit are near calls. Calls made to a unit from outside the unit are far calls. The compiler handles this transparently for you unless you're going to define things like INLINE macros or assembly language externals. Then you'd better make sure that all code is forced to be far code by bracketing all the procedure headers involved with the *{$F+}* and *{$F-}* compiler directives.

Turbo Pascal, by the way, uses the medium memory model: Near data in one 64K data segment, and far code residing in as many code segments as you need, with each unit getting its own code segment. This can get in the way if you try to declare several very large arrays. The way around Turbo Pascal's near data limitations is to use the heap and create a linked list rather than try to declare an enormous array in one piece. (The heap is wholly an artifact of the high-level language you're using and does not really involve the memory model.)

If you're working cross-language within the TopSpeed environment, you avoid trouble by making sure that all the languages involved in creating old CHARLIE.EXE are working within the same memory model. The TopSpeed languages support all models except the tiny model and the huge model. The default small model is good enough for most small projects — and certainly for getting the hang of things.

Calling All Conventions

The really ugly barrier to cross-language development, however, lies in something called "calling conventions." Like memory models, calling conventions

are sets of assumptions the compiler makes when setting up a program.

When one routine calls another routine, several things must happen: The return address must get pushed onto the stack; any parameters to be passed as part of the call must be pushed onto the stack; control must be transferred to the called routine; and finally, something must return the stack to its previous state when the called routine returns control to the caller.

These things can be done in different orders in different ways. There are two traditionally recognized calling conventions in the 86-family world:

In the Pascal calling convention, parameters are passed from left to right. In other words, given the following call:

```
Grimbler(Foo,Bar,Bas,Beep);
```

the parameter *Foo* will be pushed on the stack first, then *Bar*, then *Bas*, then *Beep*. Just before the called procedure returns control to the caller, it performs some work on the registers that causes the parameters to disappear from the stack. So by the time procedure *Grimbler* returns control to whatever called it, *Foo*, *Bar*, *Bas*, and *Beep* are simply gone, and the stack is in the same state it was before the call to *Grimbler* began.

In the C calling convention, things are pretty much the other way around. Parameters are pushed on the stack from right to left. Consider this C function:

```
fumbler(foo,bar,bas,beep);
```

Following the C calling conventions, the beep parameter goes onto the stack first, followed by bas, and then bar, and finally foo. The parameters are pushed this way so that the number of parameters passed to a C function may vary from call to call.

The sincerest hope is that when a variable number of parameters is being passed, the last parameter pushed onto the stack — and hence the only one the called procedure is certain to be able to identify using stack pointer SP — is the number of parameters passed on that particular call. The called procedure can then use this count to identify and access the remaining parameters, which lie further up the stack.

Weird? I used to think so, but it's growing on me. The problem is that the cleanup of the stack is not something that can be parametrized. The code must know how much stack space is used on each call at compile time to be able to restore the stack to its state that existed before the call was made. If the same C function can be called with three parameters at one point in

the program and with seven parameters at another point in the program, there's no way the function itself can clean up the stack. Only the code that calls the function knows at compile time how much stack space is needed for the call. Therefore, in the C calling convention, the code that calls a function takes the stack back from the called function with all the parameters still there. The caller then removes the parameters from the stack and restores the stack to its pre-call state.

These two conventions are utterly incompatible. You cannot call C code compiled using the C calling conventions from a Pascal routine compiled with the Pascal calling conventions. The tug-o-war over stack cleanup alone will send your DOS session into the bushes, regardless of parameter order.

In most systems that have allowed C and Pascal to call one another, the C code is directed (via a compiler toggle of some sort) to generate a call using the Pascal calling conventions when it calls Pascal code. Similarly, when Pascal calls a C function, that C function must have been compiled using the Pascal calling conventions. I have never yet seen a Pascal compiler that can generate calls using the C calling conventions, but I know of no reason why it couldn't be done.

JPI makes the two languages meet in the middle by creating its own calling convention, in which parameters are passed from left to right, as in Pascal, but in which the caller cleans up the stack, as in C. Furthermore, when CPU registers are available to carry parameters between caller and callee, those registers are used, making for much faster procedure calls.

The central point to be made about calling conventions is that both ends of the call must agree on the convention used. Get confused and you go bye-bye. If you're going to work cross-language, you must understand calling conventions completely. This begins by reading whatever the compiler vendor or vendors provide in the way of calling convention documentation, but the smart hacker goes in with a good debugger and watches exactly what happens — at an assembly language level — when a call is made.

Products Mentioned

TopSpeed Modula-2, V2.0
Jensen & Partners International
1101 San Antonio Road, Ste. 301
Mountain View, CA 94043
415-967-3200
Price: $199

From the Land of Lost Books

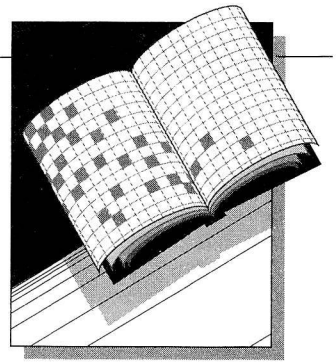
Many thanks to the people who wrote and called to say they had seen my books on the stands here and there. The bad news is that Scott, Foresman & Company was sold earlier this year to Harper & Row, which last month shut down the Scott, Foresman trade books division. When supplies are gone, that's that — my books are in limbo, I can't revert rights, and I'm a man without a publisher. So it goes with corporate megamergers.

But the programming business continues to improve. Actor 3.0 has appeared, coincident with Microsoft Windows 3.0. Modula-2 from JPI now has object extensions almost identical to those of Turbo Pascal 5.5. Stony Brook's upgraded Modula-2 and new Pascal products push the frontier of code optimization even further into the stratosphere. More on that in a future column, along with real code for some Modula-2 objects, promise . . . .

. . . drat, there's that cat again! Quick, where's my coyote call?

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 12.**



Microprocessors From the Programmer's Perspective

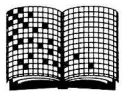
Andrew Schulman

Like everyone else, programmers generally prefer high-end machines. Many PC programmers, however, mastered instruction sets, addressing modes, and registers on the 8088 or 68000, never really coming to grips with the ins-and-outs of the fundamentally different, though backward compatible, chips such as the 80386 or 68030. Although they use 80386s, many PC programmers are still 8088 programmers at heart, possessing a surprising ignorance of high-end microprocessor architecture.

Then there are basic questions such as, What is RISC? Are the Intel 80486 and Motorola 68040 RISC chips? What is the architecture of the new IBM System/6000 family or the Sun SPARCstations? and How do you program one of those things?

To address questions like these, Robert Dewar and Matthew Smosna's *Microprocessors: A Programmer's View* provides a solid introduction to the new chips. As the subtitle indicates, this book is for programmers, not hardware design engineers; there are no descriptions of pins here, but instead, lots of code examples.

After an opening chapter on general issues — register sets, addressing modes, and instruction formats — the authors present an in-depth look at the Intel 80386 (three chapters) and Motorola 68030 (two chapters). This is followed by one chapter each on the most important Reduced Instruction Set Computer (RISC) architectures: MIPS, Sun SPARC, Intel i860, IBM (both the ROMP architecture found in the IBM RT, and the RIOS used in the new System/6000 family), and INMOS transputer.



Microprocessors: A Programmer's View

Robert B.K. Dewar and Matthew Smosna,
New York: McGraw-Hill,
1990, 462 pages, hardcover,
$39.95.

The RISC microprocessors require less explanation than the more conventional Intel or Motorola offerings, simply because the RISC chips really do have a simpler architecture. As Dewar and Smosna point out, one reason for this is that RISC manufacturers started with a clean slate while Intel and Motorola were largely driven by the need for backward compatibility.

Microprocessors also contains a lot of anecdotal material, and reflects a knowledge of the real world that is surprising in a treatise on computer architecture. One section, "A Sad Story," talks about how Intel thinks INT 5 means one thing, how IBM thinks it means another, and how this led to the IBM PCjr not using the Intel 80188 chip. Another story, which comes in the middle of a discussion of the 256-byte instruction cache on the 68020, describes a misaligned loop in Peter Norton's SI benchmarking program for the PC. A brief digression on patent law describes how DEC patented the use of the instruction pointer (IP) as a general register.

A Background in Compilers

Dewar and Smosna are both professors at the NYU Courant Institute of Mathematical Sciences. Between them, they have over 25 years experience working on compilers, including the well-respected Realia COBOL and Alsys Ada for the IBM PC, the SPITBOL compiler, and NYU's SETL language.

Why is a background in compilers useful when writing a book on the new hardware? Because, for better or for worse, most code that is run on high-end processors is generated by compilers. Thus, compiler writers determine to a large extent which features of a microprocessor are used. The fundamental observation that inspired the original RISC research was that only a small subset of the instruction set and addressing modes of most processors is commonly executed. Who better than a compiler writer to tell us which features are important and which ones, however interesting sounding, will never be used.

If an instruction exists and almost no one uses it, can its real estate on the chip be put to better use? For example, it is difficult for a compiler to take advantage of many of the niftier features of the 80386 instruction set. The 80386 has a special three-operand form of the IMUL instruction that can manipulate 64-bit integers, but there is no corresponding C data type for 64-bit integers, so the instruction is hardly ever used — even in code that needs to manipulate 64-bit integers! (In their wonderfully idiosyncratic style, Dewar and Smosna explain all this with an anecdote involving the typesetting firm owned by one of their brothers-in-law.)

So, just because a microprocessor has an instruction to work with 64-bit integers, it doesn't mean that code that handles 64-bit integers will in fact use the instruction. More likely, existing code will *never* be changed to take advantage of the new "hardware support" for 64-bit integers.

Furthermore, say the authors, code *shouldn't* often be changed to use a new feature. "Do not assume that an instruction should be used just because it is there." By way of example they provide an in-depth look at the Intel ENTER and LEAVE instructions.

"Hardware support" for a feature sounds like it should always be better than "doing it in software." Dewar and Smosna present a cogent argument that this ain't necessarily so. Again, the 80386 provides many examples. The authors discuss how "simply" by executing a FAR JMP whose target is a task state segment (TSS), one performs a 386 context switch. Sounds terrific. Why does multitasking software for the 386 ignore this magnificent hardware support, and instead do context switches in software? Because the hardware-supported context switch can easily take 300 clock cycles!

Why RISC?

This entire discussion of complex instructions that hardly anyone uses directly leads into the authors' discussion of RISC. Quoting Dan Prener of IBM, the authors note that RISC is not a reduced set of instructions, but a set of reduced instructions.

The goal of RISC is to execute one instruction per clock cycle. This requires not only simplified instructions, but also optimal use of the processor's instruction pipeline. As Dewar and Smosna explain, while a single instruction may have a latency of five clock cycles, if the instruction fetch, decode, and execution can be broken into five stages that can be overlapped with similar stages for other instructions, then average throughput can equal one instruction per clock cycle.

The key way that RISC reduces the complexity of the instruction set is by introducing a "load/store" architecture. The only instructions that interface with memory are LOAD (read memory-to-register) and STORE (write register-to-memory). All other instructions are register-to-register. This greatly reduces the number of addressing modes, which, in turn, simplifies the processor's instruction decode unit.

But a load/store architecture isn't just aesthetically pleasing. It also provides the opportunity for a potentially large performance boost. Access times for memory will always be slower than clock speeds for microprocessors. Caches only partially deal with this problem. Because a load/store architecture gates all memory access through two instructions, we can boost performance by introducing a new rule: The memory operand to a LOAD is not yet available when the instruction following the LOAD starts executing. With this rule, the processor can start executing that instruction before the read from memory has completed. The only requirement is, of course, that this next instruction not need the result of the LOAD.

The authors present an in-depth look at the Intel 80386, Motorola 68030, and each of the most important Reduced Instruction Set Computer (RISC) architectures

What then do we do with the instruction after a LOAD? Many pages in Dewar and Smosna's book are devoted to this topic. Obviously, we can insert a NOP, but then we are back where we started. In fact, a good optimizing compiler can usually reorganize code so that useful things can be done in the slot after a LOAD. But this means that good optimizing compilers are necessary to take advantage of RISC architecture. Rather than try to hide them, RISC exposes hardware features such as the speed difference between processors and memory. RISC programming means mastering the concept of software pipelining. Good compilers are needed so that most programmers will not have to remember that a LOAD from memory is a physical act that actually takes time.

The various RISC architectures explored in this book differ greatly in the extent to which they hide or expose the instruction pipeline. The MIPS chips, for example, based on the Stanford RISC research, rely on software conventions rather than hardware interlocks for handling what is called the "load delay slot." In fact, MIPS originally stood for "Microprocessor without Interlocked Pipeline Stages."

The Intel i860 is even more nontransparent. As explained by Dewar and Smosna, i860 programming looks as though it consists almost entirely of pipeline manipulation. For a good example, look in the index to Dewar and Smosna under "Breadcrumbs," and read the indicated page. (I'm not kidding!)

On the other hand, the RIOS architecture used in the IBM System/6000 sounds a lot easier to deal with. According to Dewar and Smosna, RIOS has an advantage over chips like the i860 that "with just a little knowledge of what is going on — basically little more than the rule that you should not use results you just computed — the programmer can write code that compiles in a highly efficient manner, without needing a sophisticated optimizing compiler."

The authors conclude that, "Rather than thinking of RISC as a clearly defined characteristic of microprocessors, it is better to think of RISC a being a term for a collection of design techniques used to improve performance."

Often processor's niftiest features are difficult to take advantage of in a high-level language. Chip underutilization is a major problem in our industry. It is often said that software lags behind hardware. The RISC solution is to design simpler chips. Another solution is to make heavier use of assembly language. Another solution, in the case of the underutilization of the 80386, is to use a DOS extender, so that one is using the machine as something other than a "fast XT."

Dewar and Smosna are wonderfully opinionated. Following their lengthy discussion of the rather baroque protection mechanism on the 80386, the authors forthrightly state, "The previous section is virtually incomprehensible. You probably have to read it several times to understand it, and it is still easy to get the DPLs, CPLs, and RPLs hopelessly mixed up." The title of this section is "Is All This Worthwhile?" A discussion of the insane number of addressing modes on the Motorola 68030 asks, "Had enough of this?" As computer books seem to be becoming more and more homogenized, it was a pleasure to read *Microprocessors*, a piece of technical writing which maintains the authors' voices from beginning to end.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 13.**

Listing One *(Text begins on page 127.)*

```
/* ---------------------- crypto.c ---------------------------- */

/*
 * Simple, single key file encryption/decryption filter
 * Usage: crypto keyvalue infile outfile
 */

#include <stdio.h>
#include <string.h>

void main(int argc, char *argv[])
{
    FILE *fi, *fo;
    int ct;
    char ip[8];

    char *cp1, *cp2;

    if (argc > 3)    {
        if ((fi = fopen(argv[2], "rb")) != NULL)    {
            if ((fo = fopen(argv[3], "wb")) != NULL)    {
                while ((ct = fread(ip, 1, 8, fi)) != 0)    {
                    cp1 = argv[1];
                    cp2 = ip;
                    while (*cp1 && cp1 < argv[1]+8)
                        *cp2++ ^= *cp1++;
                    fwrite(ip, 1, ct, fo);
                }
                fclose(fo);
            }
            fclose(fi);
        }
    }
}
```

End Listing One

Listing Two

```
/* -------------- des.h ---------------- */

/*
 * Header file for Data Encryption Standard algorithms
 */

/* ------- two halves of a 64-bit data block ------- */
struct LR    {
    long L;
    long R;
};

/* -------- 48-bit key permutation ------- */
struct ks    {
    char ki[6];
};

/* --------- macros to define a permutation table ---------- */
#define ps(n)       ((unsigned char)(0x80 >> (n-1)))
#define b(n,r)      ((n>r?n<r-7)?0:ps(n-(r-8)))
#define p(n)        b(n, 8),b(n,16),b(n,24),b(n,32),\
                    b(n,40),b(n,48),b(n,56),b(n,64)

/* -------------- prototypes ------------------- */
void inverse_permute(long *op, long *ip, long *tbl, int n);
void permute(long *op, long *ip, long *tbl, int n);
long f(long blk, struct ks ky);
struct ks KS(int n, char *key);

/* ---------- tables ------------ */
extern unsigned char Pmask[];
extern unsigned char IPtbl[];
extern unsigned char Etbl[];
extern unsigned char Ptbl[];
extern unsigned char stbl[8][4][16];
extern unsigned char PC1tbl[];
extern unsigned char PC2tbl[];
extern unsigned char ex6[8][2][4];
```

End Listing Two

Listing Three

```
/* ------------- encrypt.c ------------- */

/*
 * Data Encryption Standard encryption filter
 * Usage: encrypt keyvalue infile outfile
 */

#include <stdio.h>
#include "des.h"

void main(int argc, char *argv[])
{
    int i;
    struct LR op, ip;
    FILE *fi, *fo;
    struct ks keys[16];

    for (i = 0; i < 16; i++)
        keys[i] = KS(i, argv[1]);
    if (argc > 3)    {
        if ((fi = fopen(argv[2], "rb")) != NULL)    {
            if ((fo = fopen(argv[3], "wb")) != NULL)    {
```

(continued on page 148)

Listing Three *(Listing continued, text begins on page 127.)*

```
            while (fread(&ip, 1,
                    sizeof(struct LR), fi) != 0)    {
                int n;
                /* -------- initial permuation -------- */
                permute(&op.L, &ip.L, (long *)IPtbl, 64);
                /* ------ swap and key iterations ----- */
                for (n = 0; n < 16; n++)    {
                    ip.L = op.R;
                    ip.R = op.L ^ f(op.R, keys[n]);

                    op.R = ip.R;
                    op.L = ip.L;
                }
                /* ----- inverse initial permuation ---- */
                inverse_permute(&op.L, &ip.L,
                    (long *)IPtbl, 64);
                fwrite(&op, 1, sizeof(struct LR), fo);
                /* ------- to pad the last block ------- */
                ip.L = ip.R = 0;
            }
            fclose(fo);
        }
        fclose(fi);
    }
}
```

End Listing Three

Listing Four

```
/* ------------- decrypt.c ------------- */

/*
 * Data Encryption Standard encryption filter
 * Usage: decrypt keyvalue infile outfile
 */

#include <stdio.h>
#include "des.h"

void main(int argc, char *argv[])
{
    int i;
    struct LR op, ip;
    FILE *fi, *fo;
    struct ks keys[16];

    for (i = 0; i < 16; i++)
        keys[i] = KS(i, argv[1]);
    if (argc > 3)    {
        if ((fi = fopen(argv[2], "rb")) != NULL)    {
            if ((fo = fopen(argv[3], "wb")) != NULL)    {
                while (fread(&ip, 1,
                        sizeof(struct LR), fi) != 0)    {
                    int n;
                    /* -------- initial permuation -------- */
                    permute(&op.L, &ip.L, (long *)IPtbl, 64);
                    /* ------ swap and key iterations ----- */
                    for (n = 15; n >= 0; --n)    {
                        ip.R = op.L;
                        ip.L = op.R ^ f(op.L, keys[n]);

                        op.R = ip.R;
                        op.L = ip.L;
                    }
                    /* ----- inverse initial permuation ---- */
                    inverse_permute(&op.L, &ip.L,
                        (long *)IPtbl, 64);
                    fwrite(&op, 1, sizeof(struct LR), fo);
                    /* ------- to pad the last block ------- */
                    ip.L = ip.R = 0;
                }
                fclose(fo);
            }
            fclose(fi);
        }
    }
}
```

End Listing Four

Listing Five

```
/* ---------------------- des.c --------------------------- */

/*
 * Functions and tables for DES encryption and decryption
 */

#include <stdio.h>
#include "des.h"
static void rotate(unsigned char *c, int n);
static long S(struct ks ip);
static int fourbits(struct ks, int s);
static int sixbits(struct ks, int s);

/* ------- inverse permute a 64-bit string ------- */
void inverse_permute(long *op, long *ip, long *tbl, int n)
{
    int i;
    long *pt = (long *)Pmask;

   *op = *(op+1) = 0;
    for (i = 0; i < n; i++)    {
        if ((*ip & *pt) || (*(ip+1) & *(pt+1)))  {
            *op |= *tbl;
            *(op+1) |= *(tbl+1);
        }
        tbl += 2;
        pt += 2;
    }
}

/* ------- permute a 64-bit string ------- */
void permute(long *op, long *ip, long *tbl, int n)
{
    int i;
      long *pt = (long *)Pmask;

    *op = *(op+1) = 0;
    for (i = 0; i < n; i++)    {
        if ((*ip & *tbl) || (*(ip+1) & *(tbl+1))) {
            *op |= *pt;
            *(op+1) |= *(pt+1);
        }
        tbl += 2;
         pt += 2;
    }
}

/* ----- Key dependent computation function f(R,K) ----- */
long f(long blk, struct ks key)
{
    struct LR ir = {0,0};
    struct LR or;

    union    {
        struct LR f;
        struct ks kn;
    } tr = {0,0}, kr = {0,0};
    ir.L = blk;
    kr.kn = key;
    permute(&tr.f.L, &ir.L, (long *)Etbl, 48);

    tr.f.L ^= kr.f.L;
    tr.f.R ^= kr.f.R;

    ir.L = S(tr.kn);

    permute(&or.L, &ir.L, (long *)Ptbl, 32);
    return or.L;
}

/* ---- convert 48-bit block/key permutation to 32 bits ---- */
static long S(struct ks ip)
{
    int i;
    long op = 0;
    for (i = 8; i > 0; --i)    {
        long four = fourbits(ip, i);
        op |= four << ((i-1) * 4);
    }
    return op;
}
```

```
/* ------- extract a 4-bit stream from the block/key ------- */
static int fourbits(struct ks k, int s)
{
    int i = sixbits(k, s);
    int row, col;
    row = ((i >> 4) & 2) | (i & 1);
    col = (i >> 1) & 0xf;
    return stbl[8-s][row][col];
}

/* ---- extract 6-bit stream fr pos s of the  block/key ---- */
static int sixbits(struct ks k, int s)
{
    int op = 0;
    int n = (8-s);
    int i;
    for (i = 0; i < 2; i++)    {
        int off = ex6[n][i][0];
        unsigned char c = k.ki[off];
        c >>= ex6[n][i][1];
        c <<= ex6[n][i][2];
        c &=  ex6[n][i][3];
        op |= c;
    }
    return op;
}

/* ---------- DES Key Schedule (KS) function ---------- */
struct ks KS(int n, char *key)
{
    static unsigned char cd[8];
    static int its[] = {1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1};
    union    {
        struct ks kn;
        struct LR filler;
    } result;

    if (n == 0)
        permute((long *)cd, (long *) key, (long *)PC1tbl, 64);

    rotate(cd, its[n]);
    rotate(cd+4, its[n]);

    permute(&result.filler.L, (long *)cd, (long *)PC2tbl, 48);
    return result.kn;
}

/* ----- rotate a 4-byte string n positions to the left ---- */
static void rotate(unsigned char *c, int n)
{
    int i;
    unsigned j, k;
    k = *c >> (8 - n);
    for (i = 3; i >= 0; --i)    {
        j = (*(c+i) << n) + k;
        k = j >> 8;
        *(c+i) = j;
    }
}
```

End Listing Five

Listing Six

```
/* --------------- tables.c --------------- */

/*
 * tables for the DES algorithm
 */

#include "des.h"

/* --------- permutation masks ---------- */
unsigned char Pmask[] = {
    p( 1),p( 2),p( 3),p( 4),p( 5),p( 6),p( 7),p( 8),
    p( 9),p(10),p(11),p(12),p(13),p(14),p(15),p(16),
    p(17),p(18),p(19),p(20),p(21),p(22),p(23),p(24),
    p(25),p(26),p(27),p(28),p(29),p(30),p(31),p(32),
    p(33),p(34),p(35),p(36),p(37),p(38),p(39),p(40),
    p(41),p(42),p(43),p(44),p(45),p(46),p(47),p(48),
    p(49),p(50),p(51),p(52),p(53),p(54),p(55),p(56),
    p(57),p(58),p(59),p(60),p(61),p(62),p(63),p(64)
};

/* ----- initial and inverse-initial permutation table ----- */
unsigned char IPtbl[] = {
    p(58),p(50),p(42),p(34),p(26),p(18),p(10),p( 2),
    p(60),p(52),p(44),p(36),p(28),p(20),p(12),p( 4),
    p(62),p(54),p(46),p(38),p(30),p(22),p(14),p( 6),
    p(64),p(56),p(48),p(40),p(32),p(24),p(16),p( 8),
    p(57),p(49),p(41),p(33),p(25),p(17),p( 9),p( 1),
    p(59),p(51),p(43),p(35),p(27),p(19),p(11),p( 3),
    p(61),p(53),p(45),p(37),p(29),p(21),p(13),p( 5),
    p(63),p(55),p(47),p(39),p(31),p(23),p(15),p( 7)
};

/* ---------- permutation table E for f function --------- */
unsigned char Etbl[] = {
    p(32),p( 1),p( 2),p( 3),p( 4),p( 5),
    p( 4),p( 5),p( 6),p( 7),p( 8),p( 9),
    p( 8),p( 9),p(10),p(11),p(12),p(13),
    p(12),p(13),p(14),p(15),p(16),p(17),
    p(16),p(17),p(18),p(19),p(20),p(21),
    p(20),p(21),p(22),p(23),p(24),p(25),
    p(24),p(25),p(26),p(27),p(28),p(29),
    p(28),p(29),p(30),p(31),p(32),p( 1)
};

/* ---------- permutation table P for f function --------- */
unsigned char Ptbl[] = {
    p(16),p( 7),p(20),p(21),p(29),p(12),p(28),p(17),
    p( 1),p(15),p(23),p(26),p( 5),p(18),p(31),p(10),
    p( 2),p( 8),p(24),p(14),p(32),p(27),p( 3),p( 9),
    p(19),p(13),p(30),p( 6),p(22),p(11),p( 4),p(25)
};

/* --- table for converting six-bit to four-bit stream --- */
unsigned char stbl[8][4][16] = {
    /* ------------- s1 --------------- */
    14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7,
    0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8,
    4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0,
    15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13,
    /* ------------- s2 --------------- */
    15,1,8,14,6,11,3,4,9,7,2,13,12,0,5,10,
    3,13,4,7,15,2,8,14,12,0,1,10,6,9,11,5,
    0,14,7,11,10,4,13,1,5,8,12,6,9,3,2,15,
    13,8,10,1,3,15,4,2,11,6,7,12,0,5,14,9,
    /* ------------- s3 --------------- */
    10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8,
    13,7,0,9,3,4,6,10,2,8,5,14,12,11,15,1,
    13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7,
    1,10,13,0,6,9,8,7,4,15,14,3,11,5,2,12,
    /* ------------- s4 --------------- */
    7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15,
    13,8,11,5,6,15,0,3,4,7,2,12,1,10,14,9,
    10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4,
    3,15,0,6,10,1,13,8,9,4,5,11,12,7,2,14,
    /* ------------- s5 --------------- */
    2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9,
    14,11,2,12,4,7,13,1,5,0,15,10,3,9,8,6,
    4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14,
    11,8,12,7,1,14,2,13,6,15,0,9,10,4,5,3,
    /* ------------- s6 --------------- */
    12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11,
    10,15,4,2,7,12,9,5,6,1,13,14,0,11,3,8,
    9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6,
    4,3,2,12,9,5,15,10,11,14,1,7,6,0,8,13,
    /* ------------- s7 --------------- */
    4,11,2,14,15,0,8,13,3,12,9,7,5,10,6,1,
    13,0,11,7,4,9,1,10,14,3,5,12,2,15,8,6,
    1,4,11,13,12,3,7,14,10,15,6,8,0,5,9,2,
    6,11,13,8,1,4,10,7,9,5,0,15,14,2,3,12,
    /* ------------- s8 --------------- */
    13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7,
    1,15,13,8,10,3,7,4,12,5,6,11,0,14,9,2,
    7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8,
    2,1,14,7,4,10,8,13,15,12,9,0,3,5,6,11
};

/* ---- Permuted Choice 1 for Key Schedule calculation ---- */
unsigned char PC1tbl[] = {
    p(57),p(49),p(41),p(33),p(25),p(17),p( 9),p( 0),
    p( 1),p(58),p(50),p(42),p(34),p(26),p(18),p( 0),
    p(10),p( 2),p(59),p(51),p(43),p(35),p(27),p( 0),
    p(19),p(11),p( 3),p(60),p(52),p(44),p(36),p( 0),
    p(63),p(55),p(47),p(39),p(31),p(23),p(15),p( 0),
    p( 7),p(62),p(54),p(46),p(38),p(30),p(22),p( 0),
    p(14),p( 6),p(61),p(53),p(45),p(37),p(29),p( 0),
    p(21),p(13),p( 5),p(28),p(20),p(12),p( 4),p( 0)
};

/* ---- Permuted Choice 2 for Key Schedule calculation ---- */
unsigned char PC2tbl[] = {
    p(14),p(17),p(11),p(24),p( 1),p( 5),p( 3),p(28),
    p(15),p( 6),p(21),p(10),p(23),p(19),p(12),p( 4),
    p(26),p( 8),p(16),p( 7),p(27),p(20),p(13),p( 2),
    p(41),p(52),p(31),p(37),p(47),p(55),p(30),p(40),
    p(51),p(45),p(33),p(48),p(44),p(49),p(39),p(56),
    p(34),p(53),p(46),p(42),p(50),p(36),p(29),p(32)
};

/* ---- For extracting 6-bit strings from 64-bit string ---- */
unsigned char ex6[8][2][4]    = {
    /* byte, >>, <<, & */
    /* ---- s = 8  ---- */
    0,2,0,0x3f,
    0,2,0,0x3f,
    /* ---- s = 7  ---- */
    0,0,4,0x30,
    1,4,0,0x0f,
    /* ---- s = 6  ---- */
    1,0,2,0x3c,
    2,6,0,0x03,
    /* ---- s = 5  ---- */
    2,0,0,0x3f,
    2,0,0,0x3f,
    /* ---- s = 4 ---- */
    3,2,0,0x3f,
    3,2,0,0x3f,
    /* ---- s = 3 ---- */
    3,0,4,0x30,
    4,4,0,0x0f,
    /* ---- s = 2 ---- */
    4,0,2,0x3c,
    5,6,0,0x03,
    /* ---- s = 1 ---- */
    5,0,0,0x3f,
    5,0,0,0x3f
};
```

End Listings

Ray Tracing

Rendering 3-D solid objects is easier than you think

Daniel Lyke

I have always been fascinated by computer rendered three-dimensional (3-D) images, especially those that strive for realism. Various approaches exist for drawing 3-D images, ranging from algorithms for wire-frame drawings, where all objects can be seen through, and only the edges are shown, to techniques that create solid opaque objects, mirrored surfaces, shading for light sources, and shadows.

Of all the methods to render solids, ray tracing is probably the easiest to understand. Basically, you take a straight line (ray), drawn from your eye through a pixel on the screen, and see what objects it hits in the computer universe. You then find whichever object is closest to the origin of that line (your eye), and color the pixel on the screen the color of the object. If that object reflects or refracts, you just need to compute the new direction of the line and repeat the process from the new origin. Pretty simple, really. The hard part is representing the universe.

The Universe

Starting at the lowest level, a ray has an origin and a direction vector. The origin is a point in three-dimensional Cartesian space (x,y,z), and the direction is a change in each of the three axes. For the purposes of finding how far it is to an object, we'll multiply the vector by a "time" or distance.

For the purposes of this article, the objects I use are a sphere and a plane. The sphere is easy: It merely has an origin and a radius; although for simpler calculations the radius can be stored as the radius squared. The plane is a little more complicated.

Rather than express a plane as its equation in 3-D space (something such as "ax + by + cz = d"), I'll represent it as a vector normal (perpendicular) to the plane (which really means taking the "a," "b," and "c" constants in the expression above and using them as directions) and a point on the plane.

Before going further, I should mention that in the context of this article, the " · " operator (not to be confused with ".", which is the structure element selection operator in C) means multiply the individual components of the specified vector together and sum them (for example, *ray1 · ray2* results in *ray1.x * ray2.x + ray1.y * ray2.y + ray1.z * ray2.z*). Where it applies to items with multiple components, multiplication by scalars means that you multiply every component of the item by that scalar.

Finding the Intersection of a Ray and a Sphere

The sphere, in its simplest form, is expressed as the equation $x^2 + y^2 + z^2 = r^2$. This is why we store the radius

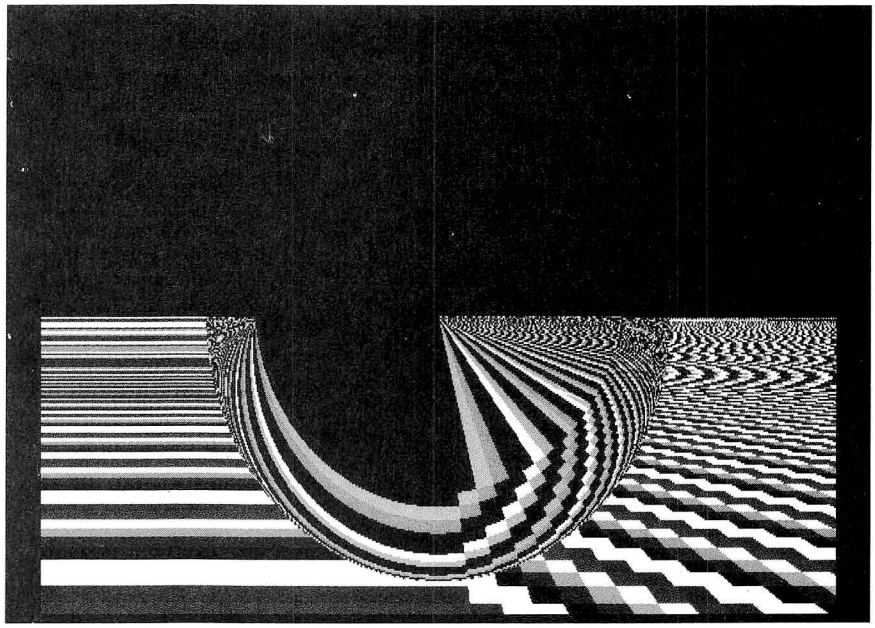


Figure 1: *Ray tracing in action*

Daniel is a programmer for Signal Data and he can be contacted at 5475 Hixson Pike, Suite I-202, Hixson, TN 37343.

as the radius squared. If we offset the ray's origin by the center of the sphere and use "time" to express where along the direction vector the ray intersects the sphere, we end up with this:

$$(time \cdot ray\_direction - ray\_origin)^2 = sphere\_radius^2$$

which, when expanded, leads to:

$$a\ time^2 + b\ time + c = 0$$

where:

$$a = ray\_direction \cdot ray\_direction$$

$$b = 2(ray\_origin - sphere\_origin) \cdot ray\_direction$$

$$c = (ray\_origin - sphere\_origin) \cdot (ray\_origin - sphere\_origin) - sphere\_radius^2$$

Delving into my old math textbooks, I come up with the equations for solving such a quadratic:

$$time = -b \pm \frac{\sqrt{B^2 - 4\ ac}}{2a}$$

We get two results because the ray can intersect the sphere at two points, when it enters the sphere and when it leaves it. There's a third possibility, which is that the argument to the square root is negative, in which case the ray never intersects the sphere at all. The function that does all of this returns the shortest positive time to intersection, or a negative number if the ray never intersects the sphere.

To find the intersection of a ray with a plane, follow this procedure:

$$a = plane\_point \cdot plane\_normal$$
$$b = plane\_normal \cdot ray\_origin$$
$$c = ray\_direction \cdot plane\_normal$$
$$time = \frac{a - b}{c}$$

Now we have the means to draw all of our objects. We merely take a ray through a screen pixel, find the time to intersect with all of our objects, take the color of the object that intersects in the shortest time, and color the pixel the color of that object. Life is nice in a non-reflecting universe. Unfortunately, interesting pictures aren't so easy. To do things like putting patterns on surfaces (such as the plane) or finding where to reflect objects that aren't flat (such as the sphere), you need to find the x, y, and z coordinates of the point of intersection by multiplying the direction of the ray by the time, and add it to the origin of the ray. To find the x coordinate use the equation:

$$x\_intersect = ray\_origin\_x + ray\_direction\_x\ (time).$$

Repeat the sequence for the y and z coordinates, respectively.

Reflections

To determine reflection, use an incident vector (the one that I'm trying to reflect) and the normal vector (the one perpendicular to the surface that we're reflecting from). You can split the incident vector into two component vectors, the sum of which is the incident vector.

The basic ray tracing technique has many applications, including video special effects and examination of architectural lighting problems

If one of these components is the normal vector, the other is the component of the vector perpendicular to the normal vector. So if you subtract this perpendicular vector from the normal vector, you end up with the reflected vector. The perpendicular vector is shown in Example 1.

If you subtract this vector perpendicular to the normal vector from the incident vector we get the normal vector and we can subtract two times the reflected vector from the incident vector to get the reflected vector, shown in Example 2.

About the Program

The program presented here compiles and runs with Zortech C++ 2.x and Flash Graphics on an EGA in 640 × 350 mode. I've hardcoded it for a specific mode so that the screen sizes could be constants (and thereby nursed a few extra clock cycles from the system) and so that I could make the colors on the sphere darker versions of the reflected colors. Listing One (page 158) is the header file RAYTRACE.HPP and Listing Two (page 158) is the C++ source. Additionally, Listing Three (page 159) lists a C version of the program.

My first version ran on a CGA in four-color mode, so it should be relatively easy to port this to anything else. The machine-specific parts are in *main( )*

and *plot( )*, with some color-specific information in *PlanePattern( )* and the defines WIDTH and HEIGHT being the screen width and height. Figure 1 illustrates the output of the program.

For ease of programming, I've put the eye at the origin (0,0,0). The screen is one unit in front of the eye: The delta (change) in the *z* component of the vector is 1. From there, calculate the position of each pixel relative to the center of the screen and put that into the *x* and *y* components of the vector.

trace( ) follows a ray through a pixel on the screen (the parameter) until it hits something, in this case through *SPHERE::Intersect( )* and *PLANE::Intersect( )*. Then it just checks the sphere and the plane to see which one has the shortest time to intersection. If both of the times are negative, the ray goes off into black sky and *trace( )* returns 0.

If we hit the plane, we need to find out where, and draw some sort of tiling pattern so that the sphere has something interesting to reflect. *PLANE::Pattern( )* takes a ray and a time to intersection and returns the appropriate color.

```
                                  incident . normal
perpendicular_vector = ----------------- normal
                                  normal . normal
```

Example 1: *Perpendicular vector*

```
                                        incident . normal
reflected_vector = incident - 2 * ----------------- normal
                                        normal . normal
```

Example 2: *Reflected vector*

Of all the methods to render solids, ray tracing is probably the easiest to understand

If the ray hits the sphere, we need to find the point of intersection with the sphere and create a ray with that as an origin, which is reflected about the normal line from the center of the sphere through that point of intersection. This is all done in *SPHERE::Reflect( )*. If the ray then hits the plane, *PLANE::Pattern( )* colors the spot appropriately. If the reflected ray does not hit the plane, then the spot is colored blue so that we can distinguish it from the black of the background.

Note that we call it with a trailing 0 *PLANE::Pattern( )* here so that these points are darker than the others.

Possible Enhancements

Before you get into the heavy work of new algorithms, some possibilities for enhancement exist. You could make the program draw the screen in low resolution, working upward fairly easily, so that the form of the image is apparent after a few of the points, and you can tell if the objects you're looking for actually show up in your viewport without rendering the entire images. Or add another pattern to the plane or another sphere (to make things simple, non-reflective).

You could also calculate several images, save them on a fast disk or in memory, and flash them quickly to the screen for animation. There's still quite a bit of precision loss at the far ends of the rays and reflections. I've got a couple of ideas on ways to solve this but I want to play with the theory a bit first. Also, there are several places where calculations could be optimized (and quite a few places where they can be done

(continued from page 154)
in parallel, given the right hardware).

Several additions suggest themselves immediately, none terribly complex. Most notable are bounded planes, light sources (for shadowing and shading), and refraction. Refraction is one of the easiest, just a modification of the reflection routine. Consult any physics textbook for the equations.

Light sources are also straightforward. Start by defining a point as a light source, and at each intersection to be displayed, trace a ray to the light source. If the ray hits nothing color the pixel as lighted, otherwise it's dark. Using angles relative to the incident ray, glossy (but nonreflective) surfaces can be handled with displays that have enough color to give the subtleties of shading. The only problem with this method is that it doesn't allow for the diffraction of light around the edges of objects (soft edged shadows have got to be faked) and prisms, lenses, and focusing mirrors don't work on the light.

To do more serious modeling, the algorithms here need to be expanded to work with bounded planes, so that more complex shapes are possible. One way to do this is to bound each plane as a convex shape with other planes. I'm still playing with ideas for clean data structures for all of this.

The basic ray tracing technique has many applications, including video special effects and examination of architectural lighting problems — special effects to commercials to architects figuring lighting problems. I'd love to see what others are coming up with and welcome your comments.

Source Code Availability

As a service to our readers, all source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Drive, Redwood City, CA 94063, or call 800-356-2002 (inside Calif.) or 800-533-4372 (outside Calif.). Specify issue number and disk format. Code is also available through M&T's Telepath on-line service (via TYMNET) and through the DDJ Forum on CompuServe (type GO DDJ).

DDJ

(Listings begin on page 158.)

Vote for your favorite feature/article.
Circle Reader Service **No. 8.**

Listing One *(Text begins on page 152.)*

```
/* RAYTRACE.HPP */
class RAY
{
   double dx, dy, dz; /* Direction vector */
   double ox, oy, oz; /* Origin */
   public:
   RAY(double x, double y, double z, double vx, double vy, double vz);
   friend class PLANE;
   friend class SPHERE;
};
class PLANE
{
   double nx, ny, nz; /* Vector normal (perpendicular) to plane */
   double px, py, pz; /* Point on plane */
   public:
   PLANE(double x, double y, double z, double vx, double vy, double vz);
   double Intersect(RAY ray);
   int Pattern(RAY ray, double time, int light);
};
class SPHERE
{
   double cx, cy, cz; /* Center of sphere */
   double r2; /* Radius squared */
   public:
   double Intersect(RAY ray);
   Reflect(RAY iray,double time, RAY &rray);
   SPHERE(double x, double y, double z, double r);
};
class VECTOR
{
   public:
   double dx, dy, dz; /* Three dimensional vector */
};
```

End Listing One

Listing Two

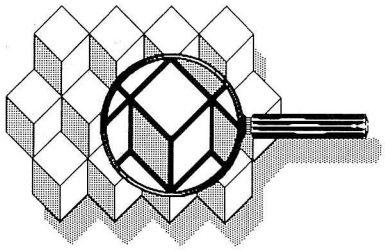
```
/* RAYTRACE.CPP */

#include <fg.h>
#include <math.h>
#include <stdio.h>
#include <conio.h>

#include "raytrace.hpp"

#define WIDTH 640
#define HEIGHT 350

inline void plot(int x,int y,int c)
{
   fg_drawdot(c,FG_MODE_SET,~0,x,y);
}

int PlanePattern(unsigned int x, unsigned int y, int light)
{
   /* Put code for different plane patterns in here */
//      return ((x + y) % 8) + 8 * light;
//      return (x % 8) ^ (y % 8) + 8 * light;
        return ((x * x + y * y) % 8) + 8 * light;
} /*----- End: PlanePattern() -----*/

RAY::RAY(double x, double y, double z, double vx, double vy, double vz)
{
   this->ox = x;
   this->oy = y;
   this->oz = z;
   this->dx = vx;
   this->dy = vy;
   this->dz = vz;
} /*----- End: RAY::RAY() -----*/

SPHERE::SPHERE(double x, double y, double z, double r)
{
   this->cx = x;
   this->cy = y;
   this->cz = z;
   this->r2 = r * r;
} /*----- End: SPHERE::SPHERE ------*/

double SPHERE::Intersect(RAY ray)
{
   double a, b, c, t1, t2, t3, close, farther;
   a = ray.dx * ray.dx + ray.dy * ray.dy + ray.dz * ray.dz;
   close = farther = -1.0;
   if(a)
      {
      b = 2.0 * ((ray.ox - this->cx) * ray.dx
         + (ray.oy - this->cy) * ray.dy
         + (ray.oz - this->cz) * ray.dz);
      c = (ray.ox - this->cx) * (ray.ox - this->cx)
        + (ray.oy - this->cy) * (ray.oy - this->cy)
        + (ray.oz - this->cz) * (ray.oz - this->cz) - this->r2;
      t1 = b * b - 4.0 * a * c;
      if(t1 > 0)
         {
         t2 = sqrt(t1);
         t3 = 2.0 * a;
         close = -(b + t2) / t3;
         farther = -(b - t2) / t3;
         }
      }
   return (double)((close < farther) ? close : farther);
} /*----- End: SPHERE::Intersect() -----*/

SPHERE::Reflect(RAY iray, double time, RAY &rray)
{
   VECTOR normal;                /* Used for readability      */
   double ndotn;                 /* Used for readability      */
   double idotn;                 /* Used for readability      */
   double idotn_div_ndotn_x2;    /* Used for optimization     */

   rray.ox = iray.dx * time + iray.ox; /* Find the point of     */
   rray.oy = iray.dy * time + iray.oy; /* intersection between  */
   rray.oz = iray.dz * time + iray.oz; /* iray and sphere.      */

   normal.dx = rray.ox - this->cx;      /* Find the ray normal  */
   normal.dy = rray.oy - this->cy;      /* to the sphere at the */
   normal.dz = rray.oz - this->cz;      /* intersection.        */

   ndotn = (normal.dx * normal.dx +
            normal.dy * normal.dy +
            normal.dz * normal.dz);
   idotn = (normal.dx * iray.dx +
            normal.dy * iray.dy +
            normal.dz * iray.dz);
   idotn_div_ndotn_x2 = (2.0 * (idotn) / ndotn);

   rray.dx = iray.dx - idotn_div_ndotn_x2 * normal.dx;
   rray.dy = iray.dy - idotn_div_ndotn_x2 * normal.dy;
   rray.dz = iray.dz - idotn_div_ndotn_x2 * normal.dz;
} /*----- End: SPHERE::Reflect() ------*/

PLANE::PLANE(double x, double y, double z, double vx, double vy, double vz)
{
   this->nx   = vx;
   this->ny = vy;
   this->nz = vz;

   this->px = x;
   this->py = y;
   this->pz = z;
} /*----- End: PLANE::PLANE() -----*/

int PLANE::Pattern(RAY ray, double time, int light)
{
   PlanePattern((unsigned)(time * ray.dz + ray.oz),
      (unsigned)(time * ray.dx + ray.ox),light);
} /*----- End: PLANE::Pattern ------*/

double PLANE::Intersect(RAY ray)
{
   double p1, p2, p3;

   p1 = this->px * this->ny + this->py * this->ny + this->pz * this->nz;
   p2 = ray.ox * this->nx + ray.oy * this->ny + ray.oz * this->nz;
   p3 = ray.dx * this->nx + ray.dy * this->ny + ray.dz * this->nz;
   return (double)((p1-p2)/p3);
} /*----- End: PLANE::Intersect() -----*/

int trace(double x, double y)
{
   static PLANE plane(-8.0, 0.0, 0.0,   0.0, 1.0, 0.001);
   static SPHERE sphere( 0.0, 0.0, 5.0, 1.0 );
   RAY ray(0.0,0.0,0.0,(x - (double)WIDTH / 2.0) *.75,
      y - (double)HEIGHT / 2.0,HEIGHT);
   double time1, time2,;
   time1 = sphere.Intersect(ray);
   time2 = plane.Intersect(ray);
   if(time1 > 0.0  && (time2 < 0.0 || time2 > time1)) /* Circle in fore */
      {
      sphere.Reflect(ray,time1,ray);
      time2 = plane.Intersect(ray);
      if(time2 > 0.0)
         {
         return plane.Pattern(ray,time2,0);
         }
      else
         {
         return 1;
         }
      }
     else if(time2 > 0.0)
      {
      return plane.Pattern(ray,time2,1);
      }
   return 0;
} /*----- End: trace() -----*/

draw()
{
   int x,y;
   for(x = 0; x < WIDTH && !kbhit(); x ++)
      {
      for(y = 0; y < HEIGHT; y ++)
         {
         plot(x,y,trace((double)x,(double)y));
         }
      }
} /*----- End: draw() -----*/

main()
{
   fg_init_egaecd();
   draw();
   getch();
   fg_term();
}
```

End Listing Two

Listing Three

```
#include <fg.h>
#include <math.h>
#include <dos.h>

#define WIDTH 640
#define HEIGHT 350
#define QUIT_OUT (!kbhit())

plot(x,y,c)
int x,y,c;
{
fg_drawdot(c,FG_MODE_SET,~0,x,HEIGHT - y);
}

typedef struct S_RAY
{
        double dx, dy, dz; /* Direction vector */
        double ox, oy, oz; /* Origin */
} RAY;

typedef struct S_PLANE
{
        double nx, ny, nz; /* Vector normal (perpendicular) to plane */
        double px, py, pz; /* Point on plane */
} PLANE;

typedef struct S_SPHERE
{
        double cx, cy, cz; /* Center of sphere */
        double r2; /* Radius squared */
} SPHERE;

typedef struct S_VECTOR
{
        double dx, dy, dz; /* Three dimensional vector */
} VECTOR;

double sphere_intersect(RAY, SPHERE);
double plane_intersect(RAY, PLANE);
void reflect(VECTOR *, VECTOR *, VECTOR *);
double sphere_intersect(RAY ray, SPHERE sphere)
{
        double a, b, c, t1, t2, t3, close, farther;
        a = ray.dx * ray.dx + ray.dy * ray.dy + ray.dz * ray.dz;
        close = farther = -1.0;
        if(a)
                {
                b = 2.0 * ((ray.ox - sphere.cx) * ray.dx
                  + (ray.oy - sphere.cy) * ray.dy
                  + (ray.oz - sphere.cz) * ray.dz);
                c = (ray.ox - sphere.cx) * (ray.ox - sphere.cx)
                  + (ray.oy - sphere.cy) * (ray.oy - sphere.cy)
                  + (ray.oz - sphere.cz) * (ray.oz - sphere.cz) - sphere.r2;
                t1 = b * b - 4.0 * a * c;
                if(t1 > 0)
                  {
                  t2 = sqrt(t1);
                  t3 = 2.0 * a;
                  close = -(b + t2) / t3;
                  farther = -(b - t2) / t3;
                  }
                }
        return (double)((close < farther) ? close : farther);
} /*----- End: sphere_intersect() -----*/

double plane_intersect(RAY ray, PLANE plane)
{
        double p1, p2, p3;
        p1 = plane.px * plane.ny + plane.py * plane.ny + plane.pz * plane.nz;
        p2 = ray.ox * plane.nx + ray.oy * plane.ny + ray.oz * plane.nz;
        p3 = ray.dx * plane.nx + ray.dy * plane.ny + ray.dz * plane.nz;
        return (double)((p1-p2)/p3);
} /*----- End: plane_intersect() -----*/

void reflect(VECTOR *normal, VECTOR *incident, VECTOR *r)
{
        double ndotn, idotn;
        ndotn = (normal->dx * normal->dx +
                                normal->dy * normal->dy +
                                normal->dz * normal->dz);
        idotn = (normal->dx * incident->dx +
                                normal->dy * incident->dy +
                                normal->dz * incident->dz);
        r->dx = incident->dx - (2.0 * (idotn) / ndotn) * normal->dx;
        r->dy = incident->dy - (2.0 * (idotn) / ndotn) * normal->dy;
        r->dz = incident->dz - (2.0 * (idotn) / ndotn) * normal->dz;
} /*----- End: reflect() -----*/

int plane_pattern(int x, int y, int light)
{
        return ((x + 16384) % 8) ^ ((y + 16384) % 8) + 8 * light;
} /*----- End: plane_pattern() -----*/

int trace(int x, int y)
{
        static PLANE plane = { 0.0, 1.0, 0.001, -8.0, 0.0, 0.0};
        static SPHERE sphere = { 0.0, 0.0, 5.0, 9.0 };
        VECTOR v1, v2, v3;
        RAY ray;
        double t1, t2, time;
        ray.ox = 0.0;            /* Set the ray origin to the eye */
        ray.oy = 0.0;
        ray.oz = 0.0;
        ray.dz = 1.0;           /* Set the direction through the pixel  */
        ray.dy = -((double)y - (double)HEIGHT / 2.0) / 100;
        ray.dx = ((double)x - (double)WIDTH / 2.0) / 120;

        t1 = sphere_intersect(ray,sphere);
        t2 = plane_intersect(ray,plane);
        if(t1 > 0.0 && (t2 < 0.0 || t2 > t1)) /* Circle in fore */
          {
          v1.dx = ray.dx; v1.dy = ray.dy; v1.dz = ray.dz;
          v2.dx = ((ray.dx * t1 + ray.ox) - sphere.cx);
          v2.dy = ((ray.dy * t1 + ray.oy) - sphere.cy);
          v2.dz = ((ray.dz * t1 + ray.oz) - sphere.cz);
          reflect(&v2,&v1, &v3);
          ray.ox += ray.dx * t1; ray.oy += ray.dy * t1; ray.oz += ray.dz * t1;
          ray.dx = v3.dx; ray.dy = v3.dy; ray.dz = v3.dz;
          t2 = plane_intersect(ray,plane);
          if(t2 > 0.0)
                {
                return plane_pattern((int)(t2 * ray.dz + ray.oz),(int)(t2 *
                                                        ray.dx + ray.ox),0);
                }
          else
                {
                return 1;
                }
          }
        else if(t2 > 0.0)
          {
          return plane_pattern((int)(t2 * ray.dz + ray.oz),(int)(t2 *
                                                        ray.dx + ray.ox),1);
          }
        return 0;
} /*----- End: trace() -----*/

draw()
{
        int x,y;
        for(x=0;x< WIDTH && QUIT_OUT; x++)
                {
                for(y = 0; y < HEIGHT; y++)
                        {
                        plot(x,y,trace(x,y));
                        }
                }
} /*----- End: draw() -----*/

main()
{
        fg_init_all();
        draw();
        while(QUIT_OUT);
        getch();
        fg_term();
}
```

End Listings



The folks at **Hyperkinetix** have released Version 1.21 of The Builder, a batch file compiler and language extender that includes menus as command syntax. *DDJ* met with Hyperkinetix's "master codeblaster," Tom Campbell, who demonstrated Builder's simplicity and ease of use. Tom combined aspects he liked in C, Basic, and Hypertalk; thus Builder is English-like yet efficient — Tom wrote a simple menu shell program with four or five lines of Builder code.

You can create custom menus and distribute them freely, rather than purchase canned programs for every machine you need menus for. You can hide details of a program so that users cannot tamper with it, filter out unwanted keystrokes, and display boxes, colored text, and position the cursor directly (without using ANSI.SYS).

Builder imports .BAT files and translates them into compiled .COM or .EXE programs. All batch commands are duplicated, and structures such as *Case, While,* and *Repeat* and functions such as *DiskReady* and *Input* are included. *DiskFree* determines the amount of available disk space, *RenSub* renames subdirectories, *FileSize* returns the size of a file in bytes, *ReadLine* and *WriteLine* give full I/O capabilities, and *GetKey* processes keystrokes, including function keys. Builder's variable types include integer and *LongInt,* and subroutines allow you to create your own keywords. You can build PopUp or DropDown menus that act as block structure statements, and Builder programs automatically use a Microsoft-compatible mouse if one is present. Sells for $149.95, and includes a money-back guarantee and telephone, BBS, and CompuServe tech support. Reader service no. 20.
Hyperkinetix, Inc.
666 West Baker, Ste. 405
Costa Mesa, CA 92626
714-668-9234

CodeTAP 386 is a source-level, run-time debugging tool for 80386 embedded systems, from **Applied Microsystems Corp**. It provides a transparent window into the internal functioning of the 80386 while executing code in the target environment, using emulation technology integrated in a custom chip. Developers currently use native debuggers and software monitors for debugging, and use emulators for full system integration and for solving real-time problems. CodeTAP bridges these technologies.

A major difference between CodeTAP and software monitors is CodeTAP's ability to monitor and control code execution in the target without utilizing target memory, I/O, or requiring prior code modification — it is electrically transparent in the target, allowing engineers to single step or operate at full clock speeds up to 33 MHz with no wait states.

A source-level debugger is included for both high-level language and assembly debugging. Combined with Phar Lap's 386 ASM/LinkLoc, the debugger software supports Intel OMF-compatible languages, Microsoft C, and other popular compilers.

The company claims CodeTAP's greatest asset is that it can shorten development time for embedded systems. Pricing begins at $5,000. Reader service no. 21.
Applied Microsystems Corporation
5020 148th Ave. NE
P.O. Box 97002
Redmond, WA 98073-9702
206-882-2000

Stony Brook Software has announced Version 2.1 of their Modula-2 Professional system and QuickMod compilers. Additional code optimization includes loop rewriting, register parameter passing, and automatic inline expansion of procedures. Also new is Presentation Manager support, enhanced MS Windows support, an integrated execution profiler, and an integrated object librarian. All components are integrated under a single environment.The Professional system includes the Stony Brook environment, the QuickMod and Optimizing compilers, the debugger, and the run-time library. The price is $295. Reader service no. 22.
Stony Brook Software
187 E. Wilbur Rd., Ste. 9
Thousand Oaks, CA 91360
805-496-5837

Programmers who like the structure of Modula-2 and need the processing power transputers offer now have the option of using Modula-2 instead of C or Occam. **Computer System Architects** has developed a Modula-2 package based on the third edition of Wirth's *Programming in Modula-2. DDJ* spoke with Richard Ohran, who ported the compiler to the transputer and who worked with Wirth on the original implementation on the Lilith at the Swiss Federal Institute of Technology. He said that "our version of Modula has features that facilitate using more than one processor. It is set up to replicate a copy of itself into another transputer." The full symbolic network debugger identifies deadlocks and run-time errors, and can display the state of every process in the net regardless of distance from the host processor.

Graphical user interface routines support overlapping windows, menus, down-loadable fonts, and mouse-driven cursor inputs. The compiler operates in two passes and generates binary object code with optimized relative addressing offsets. Extensions allow generation of all transputer parallel processing constructs.

The transputer library is a collection of precompiled Modula-2 procedures that can be imported into your programs. The transputer network debugger can thread its way into a deadlocked net of transputers and display the state of each task within each processor until it finds the cause. It then displays the symbolic notation and shows the position and names, types, and contents of all data structures. The system sells for $995, and the source code to the Medos system kernel (which allows programs to call other programs as overlays that link dynamically to resident modules and so inherit characteristics from the program environment they are called into) and the library modules is another $1,000. Discounts available to educational institutions. Reader service no. 23.
Computer System Architects
950 N. University Ave.
Provo, UT 84604-3422
801-374-2300

Coleman Softlabs has released Overlay Designer, which is "computer-aided design" for programmers using Plink86 and RTLink to design overlay structures. If you are building a product with more than 50 overlays, Overlay Designer can apparently save you a lot of time. Mike Milsner of Symantec told *DDJ* that it "saved me hours and days of work. I like it a lot. We use Plink, and have an overlay structure that grew and grew — to more than 170 modules. Instead of mapping the structure by hand I used Overlay Designer and it automatically prints it out and lets me know if two things are in memory at the same time. It helps to determine what goes into the root and which small pieces can be overlayed on disk, so the user can ping-pong between, say, a spreadsheet and a graph and not

(continued on page 166)

(continued from page 161)
have the system crash."

Overlay Designer reads the MAP file output by the linker to get the information necessary to build its database. Overlay Designer's graphic, algorithmic, and tabulation tools help you refine the overlay structure. The on-screen interaction includes a report that shows direct "equal level overlay" conflicting calls, 35 dynamic tables to cross-reference public symbol usage, ability to move code between overlays and instantaneously compute the result, and tables to show program sizes by file, module, and overlay. Price: $595. Reader service no. 24.
Coleman Softlabs, Inc.
296 Bay Rd.
Atherton, CA 94027
415-322-9006

A full standard Common Lisp for Unix System V and X-Window-based multiprocessing systems is available from **Top Level**. TopCL Version 2.0 includes a foreign function interface for calling C and Fortran code, requires minimal recoding of existing algorithms and libraries, and provides debugging tools not available with strictly serial programming languages. The company claims that TopCL provides near-linear, linear, and better than linear performance speedup, that synchronization is handled implicitly, and that applications written in TopCL can run as fast as the multiprocessors and memory bus hardware will allow. TopCL is ANSI-compatible. Reader service no. 25.
Top Level, Inc.
196 N. Pleasant St.
Amherst, MA 01002
413-256-6405

Version 4 of the source code analysis tool for C, PC-lint, has been released by **Gimpel Software**. PC-lint runs under MS-DOS or OS/2, and analyzes C programs for bugs and inconsistencies. PC-lint looks across multiple modules and helps make programs more maintainable. It is also useful for porting programs to new machines, operating systems, compilers, and memory models.

Kathy Bell of Softcraft told *DDJ* that "we use it all the time and like it very much. We have a large project (125 source files, 2.5 Mbytes of code, 65 header files) and have been able to lint all of our source files at one time. The atest version is the best so far. It tells us which header files we don't need and which functions aren't being called. With projects this large, you forget what you worked on a year ago, and with several people working on a project, it is very helpful to know these things."

New diagnostics include checks for compile-time objects such as macros, typedefs, and declarations; PC-lint reports if they are not used locally and globally. A lint object module captures in binary form all the external information of a module, allowing for incremental linting via a make file. New options customize message suppression, Unix-style options are supported, and error messages give detailed information regarding files, type differences, precision loss, and so on. And you can be alerted to silent conversions caused by prototypes. PC-lint requires a minimum of 196 Kbytes of memory, and will use all available memory. It sells for $139. Reader service no. 29.
Gimpel Software
3207 Hogarth Ln.
Collegeville, PA 19426
215-584-4261

Multitasking for C programs is possible with the DIVVY software package from **Drumlin Inc.**, which provides a way to divide up processor time. DIVVY allows control operations to be programmed separately, and causes the CPU to continually switch between tasks at run time, effectively running all tasks simultaneously. DOS is inherently single-tasking; DIVVY allows a multitasking application to be self-contained, avoiding the need to upgrade to OS/2 or Unix to achieve multitasking. Structured as a library of routines that can be linked into programs written in MS C or Turbo C, DIVVY can support an unlimited number of tasks, flags, and queues. Standard libraries and DOS calls are usable through the scheduler system, which operates on a priority-based, non-preemptive basis. The package sells for $229. Reader service no. 26.
Drumlin, Inc.
1011 Grand Central Ave.
Glendale, CA 91201
818-244-4600

FreeForm, a C, symbolic, source-level debugger for ROMable 68000-based applications, is new from **Software Development Systems**. The company claims this product will deliver the highest level of symbolic debugging without the use of windows or an in-circuit emulator. FreeForm is command-language driven, allowing you to choose precisely which code and data are being displayed at any given moment. You can plant breakpoints at C-level source statements, inspect and modify C symbols, and view special objects such as stacks or symbol sets. If an object is too big to fit on a screen, it can be redirected to a file and edited. Arrays, enumerations, and structures can be displayed in full symbolic form, and complex data structures such as linked lists and trees can be displayed automatically — the debugger assumes that the entire screen is available for display.

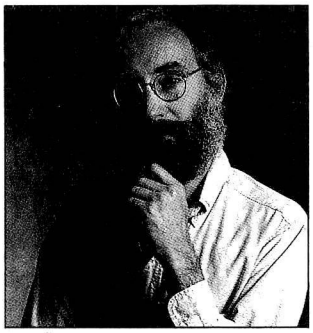
FreeForm controls the target application through available serial or I/O ports by communicating with a target monitor program that can be configured to operate on virtually all 68000-family targets. Prices range from $1,795 to $3,595, depending on the host machine. Reader service no. 27.
Software Development Systems, Inc.
4248 Belle Aire Ln.
Downers Grove, IL 60515
800-448-7733

Watcom has announced both their Fortran 77/386 and C/386 Optimizing Compilers and Tools Version 8.0. F77/386 V8.0 supports the development of Fortran programs that run in 32-bit protected mode on the 386, typically with a DOS extender. F77/386 and Watcom C8.0/386 share code optimization technology and support tools such as the debugger, linker, profiler, and object librarian. Both are run-time compatible, allowing inter-language function calls. List price for F77/386 V8.0 is $1,295; the Professional Edition of C8.0/386 is $1,295, Standard Edition is $895. Reader service no. 30.
Watcom
415 Phillip St.
Waterloo, Ontario
Canada N2L 3X2
519-886-3700

Lahey Computer Systems announces Version 3.0 of F77L-EM/32, their 32-bit DOS Fortran compiler, which combined with the Ergo OS/386 DOS extender lets users exceed the DOS 640K barrier and use extended memory. You must purchase the Ergo OS/386 extender ($395) in order to use the Lahey compiler. Virtual memory and run-time licenses are included in the $395. The compiler is compatible with DESQview, so you can multitask applications and run other applications while compiling in the background. Version 3.0 includes a new editor and *make* utility. If you use a 386, you need an Intel or Weitek math coprocessor, 1 Mbyte of extended memory, and DOS 3.x or greater. You can also use it on a 486. The compiler retails for $895. Reader service no. 31.
Lahey Computer Systems
P.O. Box 6091
Incline Village, NV 89450
702-831-2500

DDJ

Thoughts on the Reviewing of Software



I think that the practice of sending out "reviewer's guidelines" along with review copies of a complex product is a good one. Of course such guidelines are biased — slanted toward the features of the product that the vendor thinks are most likely to impress the reviewer, engineered to draw the reviewer's attention away from the aspects of which the vendor isn't so proud. But I believe that they also serve two commendable purposes.

First, they tell the reviewer what the vendor had in mind in developing the product. This is sometimes far from clear. Reviewer's guidelines say, in effect, "Here's what we think we've accomplished; here are the features on which we want to be judged." With some products, it really does help a reviewer to know what the company had in mind; consider, for example, B&E Software's RagTime, an integrated software package for the Macintosh.

B&E is now marketing the third version of RagTime in the U.S. The European company faces the negative impression created by two less-than-impressive attempts to sell the product here through intermediaries. It faces the challenge of selling an integrated do-all application for the Macintosh at a time when Apple is pushing the idea of small, tool-like applications that share functionality. And it faces the problem of any integrated package: None of its included applications is the best in its class. If RagTime gets reviewed on the basis of past perceptions, or of Apple's vision of the future, or of its components, it will not fare well. Reviewer's guidelines would let B&E define the terms on which it wants RagTime to be evaluated.

The other virtue of reviewer's guidelines is that they virtually force marketing and engineering to communicate. As we have all seen demonstrated, a press release or advertisement can be written with almost no knowledge of the product. Not so with reviewer's guidelines. Before you start suggesting to reviewers what sort of benchmarks are appropriate in your product's category, you'd better know something about the product's performance.

I used the term "functionality" above with some reservations. When I was at *InfoWorld* in 1981, we argued at some length about whether or not there was such a word. We were not alone in disliking the word: Ted Nelson fumes about it in *Computer Lib.* But the word does exist and is used, because it's needed. It's ugly, but it's right.

I wonder, though, if software reviews today are using all the right words. In 1974, features and performance were preeminent. Design and depth are arguably more important today, but those words don't often appear as subheads in reviews.

Trip Hawkins of Electronic Arts has spoken eloquently about the need for software to be deep: Capable of being used for years, with new folds to discover, and thus a long shelf life. Naive users have different needs from experienced users, and a software product ought to have value for both groups. HyperCard is an extraordinary case study in depth: All the way from cut-and-paste application generation to the brink of writing utilities in C. No, HyperCard doesn't teach C, but everyone who gets deeply into HyperCard is strongly tempted to write an external command, and HyperCard's external command interface takes away a lot of the pain of learning to program the Mac.

Perhaps reviewers ought to give more attention to such subjective issues as depth and design, and less to those gargantuan tables of features and benchmark results.

Then again, objectivity is safer than subjectivity.

On June 21 of this year, the Supreme Court removed the opinion defense in libel suits. Writers of newspaper or magazine opinion columns or of letters to the editor, cartoonists, restaurant and software reviewers, can all be sued for libel if the opinions they express cause damage to someone. Negative software reviews, it goes without saying, cause damage to the company whose product is reviewed. The plaintiff has to prove that the damaging statement was not true, but legal experts predict that a lot of suits that would never before have been considered will now not only be filed, but will actually go to trial.

The same legal experts predict that this decision will cause commentators and reviewers to be much less outspoken in their views. The software reviewers of my acquaintance do not seem to be modifying their behavior in response to the ruling. They may be sorry.

Under the circumstances, I think I'll reserve my opinion of the present Supreme Court justices. I don't think we could print it anyway.

Michael Swaine

Michael Swaine
editor-at-large

#169 OCTOBER 1990 $3.50 ($4.50 CANADA)

15 YEARS OF POWER PROGRAMMING

Dr. Dobb's JOURNAL

SOFTWARE TOOLS PROFESSIONAL PROGRAMMER

ANNUAL OPERATING SYSTEMS ISSUE

Breaking the 640K Barrier

YOUR OWN DOS EXTENDER!

PLUS

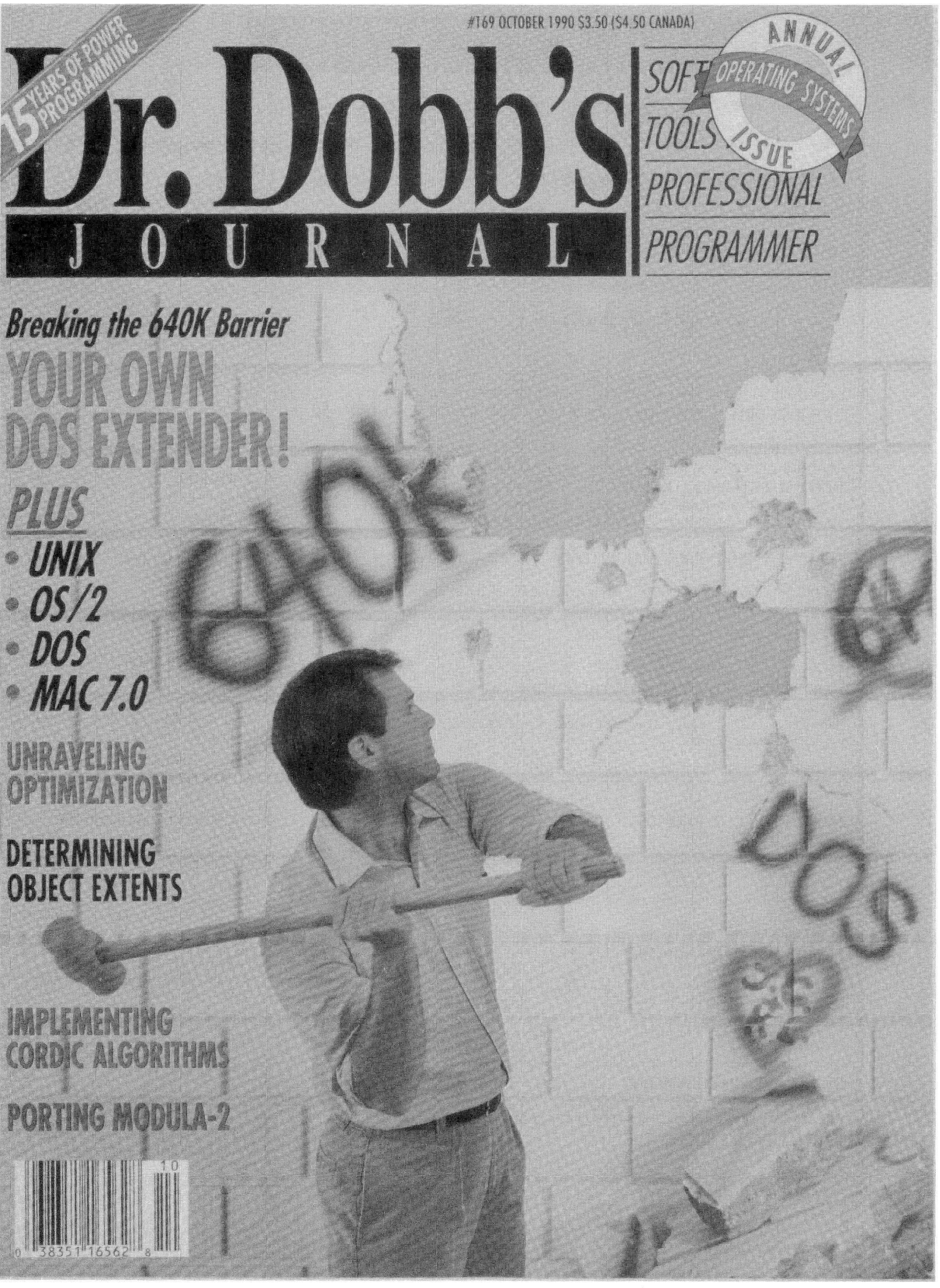
- UNIX
- OS/2
- DOS
- MAC 7.0

UNRAVELING OPTIMIZATION

DETERMINING OBJECT EXTENTS

IMPLEMENTING CORDIC ALGORITHMS

PORTING MODULA-2



0 38351 16562 8 10

CONTENTS

OCTOBER 1990
VOLUME 15, ISSUE 10

FEATURES

ROLL YOUR OWN DOS EXTENDER: PART I 16
by Al Williams
In this first installment of a two-part article, Al shares a DOS extender called PROT and takes a look at protected-mode programming.

OPENING OS/2's BACKDOOR 28
by Andrew Schulman
The good news is that protected-mode OS/2 prevents task clashes by taking control of memory. The bad news is this makes it hard to perform memory-mapped I/O and write diagnostic tools. Andrew's generic device driver helps solve these problems.

CLOSING DOS's BACKDOOR 42
by John Switzer
Accessing DOS without going through the INT21 function handler can have serious repercussions. John shows how to close these vulnerable backdoors.

RAM DISK DRIVER FOR UNIX 50
by Jeff Reagen
Jeff's RAM disk lets you more efficiently manage */tmp* files, reduce overhead associated with file loading, and increase performance by using RAM as a primary swap device.

OPTIMAL DETERMINATION OF OBJECT EXTENTS 58
by Victor J. Duvanenko, Ronald S. Gyurcsik, and W.E. Robbins
Here's an algorithm for efficiently determining the minimum and maximum reaches of 2-D and 3-D graphical objects.

IMPLEMENTING CORDIC ALGORITHMS 152
by Pitts Jarvis
CORDIC (COordinate, Rotation, DIgital Computer) algorithms let you use one core routine to compute sines, cosines, exponentials, logarithms, and other transcendentals.

EXAMINING ROOM

UNRAVELING OPTIMIZATION IN MICROSOFT C 6.0 62
by Bruce D. Schatzman
Bruce investigates the practical and theoretical aspects of code optimization, using Microsoft C 6.0 as his example platform.

PROGRAMMER'S WORKBENCH

KERMIT FOR OS/2: PART II 72
by Brian R. Anderson
While wrapping up his Kermit port, Brian examines the communications capabilities of OS/2, along with low-level screen and video I/O.

COLUMNS

PROGRAMMING PARADIGMS 123
by Michael Swaine
Michael reports on the recent MacWorld conference, focusing on System 7 developers and developments.

C PROGRAMMING 131
by Al Stevens
Al catches up on his correspondence and follows up on his token-pasting.

STRUCTURED PROGRAMMING 139
by Jeff Duntemann
Jeff returns to Zeller's Congruence before peering through the open window of Windows 3.0.

PROGRAMMER'S BOOKSHELF 145
by Ray Duncan
Do software aficionados need to know about computer architectures? "Yes," says Ray, and the recently published *Computer Architecture* is a good place to start.



DEPARTMENTS

EDITORIAL 6
by Jonathan Erickson

LETTERS 8
by you

SWAINE'S FLAMES 168
by Michael Swaine

PROGRAMMER'S SERVICES

ADVERTISER INDEX 160
where to go for more information on products

OF INTEREST 161
compiled *by Janna Custer*

PROGRAMMER'S MARKETPLACE 162
classified ads

SOURCE CODE AVAILABILITY

As a service to our readers, all source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (inside Calif.) or 800-533-4372 (outside Calif.). Specify issue number and disk format. Code is also available through the DDJ Forum on CompuServe (type *GO DDJ*) and through M&T's Telepath Conferencing system.

NEXT ISSUE

November brings our annual object-oriented programming issue, where we'll examine programming problems — and present practical solutions — in a variety of OOP languages.

Courting Trouble

In the most recent cat fight between Intel and Advanced Micro Devices (AMD), lawyers for both sides claimed victory, at least for the time being. In his preliminary injunction, Judge William Ingram said AMD could continue to use a product numbering system that mirrors Intel's product line, but backed off from permitting AMD use of the name "Intel" in its advertisements.

This case revolves around a copyright infringement lawsuit launched by Intel over a supposedly Intel-compatible 80287 math coprocessor AMD began selling in April. The judge says AMD can't boast anymore that its 80287 is the 100 percent equivalent of Intel's although, as an AMD spokesman said, the AMD chip is "a reverse-engineered coprocessor that incorporates Intel microcode," microcode AMD received from Intel in a 1976 patent exchange agreement.

But what's at stake here isn't just the "80287" label — that fish isn't big enough to fry. The big perch in the pond is the 80386, potentially one of the most profitable chips around. What's going on is that Intel is setting the stage to protect the name "80386" against chip cloning by AMD and other manufacturers. While Intel has apparently conceded its right to names like "8086," "80286," and "80287," the company is claiming ownership of and protection for the name "80386" (and "80486," etc.). As you can expect, that doesn't set well with chip manufacturers who'd like to cash in with compatible chips. AMD's response is that the "80*x*86" family name has become generic, somewhat like the term "PC," I guess.

By the time you're reading this, the situation might have taken another faltering step towards settlement because of a September ruling on the *other* Intel/AMD rhubarb, the "second source" dispute. To briefly recap *that* story: When IBM decided on Intel's 80*x*86 architecture as the CPU for its microcomputers, Big Blue stipulated that Intel had to provide a second manufacturing source for the chips. Consequently, Intel licensed the 80*x*86 architecture to other chip vendors, usually by means of technology exchange agreements. After that it was business as usual, until the 80386. Intel, recognizing a golden goose when it saw one, refused to allow AMD to manufacture the chip. Since the original agreement between the two companies said that disputes would be settled by arbitration, not legal action, AMD tried to get Intel to a mediator. Intel refused. AMD then went to court just to get to arbitration.

Who's right? It depends on which court case you're talking about. As for the microcode copyright ruling, I don't question Intel's right to protect products it produces and the names it uses to identify them. However, it also seems there may be some First Amendment questions involving AMD's right to say in advertisements (or otherwise) whether or not their products are "Intel compatible." The whole question of second sourcing, on the other hand, will likely drag on and I expect the windup will hinge on the exact wording of the agreements *and* whether or not those agreements are valid and binding. In any event, it will be the judge (or judges) who eventually decides, not you or me.

What I do know is that I first wrote about these legal calisthenics several years ago. Not only have the issues *not* been resolved, but new lawsuits keep getting pushed onto the stack: Look-and-feel, patents, copyright, and on and on.

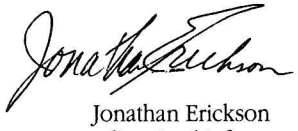
To keep all this in perspective, however, note that when it comes to litigious propensities, the computer industry doesn't stand alone. A case in point: Kellogg's, those folks who put breakfast cereal on your table every morning, have launched a legal salvo at arch rival Nabisco over, of all things, shredded wheat. Each company claims "nutritional superiority" and says the other company is copying the other's product features and marketing strategy. Sound familiar? At least some good could come out of the Kellogg/Nabisco suit — we might get more better fiber.

Better yet, maybe we all should join the League for Programming Freedom for a few bars of the "Hexadecimal Chant" (loosely sung to the tune of Country Joe MacDonald's "Vietnam Rag"):

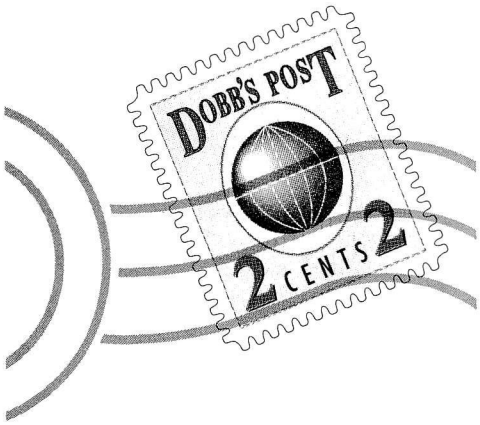
1-2-3-4, Kick that lawsuit out the door,
5-6-7-8, Innovate, don't litigate,
9-A-B-C, Interfaces should be free,
D-E-F-0, Look-and-feel has got to go.

The Official *DDJ* Bookmark

Finally, in our never ending quest to bring you practical tools for everyday use, we're including this month the official *DDJ* bookmark. You'll find it attached to a tear-on-the-dotted-line card accompanying the "Programmer's Bookshelf" column on page 145. Counter to the prevailing legal winds, this bookmark is provided license-free; copy it and pass it around to your friends. As you might expect, technical support is minimal. If you have any questions, call my lawyer.



Jonathan Erickson
editor-in-chief



Still Going in Circles

Dear *DDJ*,

As usual, the July 1990 issue proved to me that my *DDJ* subscription is worth every penny. I enjoyed all the articles, including Mr. Paterson's on drawing circles, but I'd like to add my two cents on the subject, nonetheless.

Mr. Paterson's analysis of the circle drawing problem using calculus was interesting, but there are some simplifications overlooked which can be found if one applies some algebra and a little elbow grease. Allow me to borrow from Mr. Zigon, Mr. Lee, and Mr. Paterson, and state the problem as follows. Given a point (x,y) known to be near the circle, find the next contiguous point. Furthermore, we will confine ourselves to points in the octant defined by $x >= 0$, $y <= r$, $x <= y$.

Since we are only considering points on the octant from $(0, r)$ traveling clockwise, the next point will either be $(x+1, y)$ or $(x+1, y-1)$. The problem then becomes to find which is closer to a perfect circle. We know that if (x,y) is on the circle, $(x+1,y)$ is above or on the circle and $(x+1,y-1)$ is below or on the circle. I won't prove this, but a drawing should be convincing enough. If the distance from the circle to $(x+1,y)$ is greater than the distance from $(x+1,y-1)$ to the circle, then we will choose $(x+1,y-1)$, otherwise we will choose $(x+1,y)$ as the next point. If the two distances are identical, we'll arbitrarily choose $(x+1,y)$ as the next point.

Let $e(x,y)$ be the error function, which is the distance from a perfect circle of radius r centered on the origin to the point (x,y). $e(x,y)$ is expressed as $e(x,y) = x^2 + y^2 - r^2$.

$e( )$ can be easily derived by using the definition of a circle and Pythagoras' Theorem. At this point, the circle drawing algorithm looks like Example 1. The goal now is to reduce the amount of computation needed in the loop. Most of it is being performed in the *if* statement. The problem is the inequality which computes $e( )$ twice:

$$e(x+1,y) > -e(x+1,y+1)$$

Algebra to the rescue. First, expand the right-hand side, and substitute using the definition of $e( )$.

$$e(x+1, y-1) = (x+1)^2 + (y-1)^2 - r^2$$
$$e(x+1,y-1) = (x+1)^2 + y^2 - 2y+1 - r^2$$
$$e(x+1,y-1) = e(x+1,y) - 2y+1$$

Substitute into the original inequality:

$$e(x+1,y) > -e(x+1,y)+2y-1$$

Add $e(x+1,y) - 2y+1$ to both sides:

$$2e(x+1, y) - 2y + 1$$

Divide by 2:

$$e(x+1,y) - y>0$$

Note that the 1/2 drops out, since we are performing an integer comparison. Now $e( )$ is computed only once instead of twice each time through the loop. The algorithm now looks like Example 2.

Now, to get rid of the remaining $e( )$. The above expressions showed that $e(x+1,y-1)$ can be expressed in terms of $e(x+1,y)$. If $e(x+1,y)$ can be expressed in terms of $e(x,y)$, the calculation of $e( )$ can be written in an iterative form (each new value of $e( )$ computed by modifying the previous $e( )$), which is (hopefully) easier to compute.

$$e(x+1,y) = (x+1)^2 + y^2 - r^2$$
$$e(x+1,y) = x^2 + 2x+1+y^2 - r^2$$
$$e(x+1,y) = e(x,y)+2x+1$$

Adding a variable, e, to accumulate $e( )-y$ and changing the multiplication by 2 to a left shift by 1 yields the final version shown in Example 3.

```
y =r ;
x =0 ;
while (x <= y) {
  plot8 (x, y) ;
  if (e(x + 1, y) > -e(x + 1, y -1))
        y-- ;
  x++ ;
}
```

Example 1

```
y = r ;
x = 0 ;
while (x ,= y) {
     plot8 (x, y) ;
     if (e (x + 1, y) - y > 0)
          y--;
     x++;
```

Example 2

```
y = r ;
x = 0 ;
e = -y ;       /* e(o, r) == 0, since
this point is on the circle */
while (x <= y) {
     plot8 (x, y) ;
     e += (x << 1) + 1 ;
     if (e > 0) {
                    y-- ;
                    e -= (y << 1) ;
          }
             x++ ;
}
```

Example 3

There's my version of the circle plotting algorithms. Plotting a circle of radius 1,000,000 took 3.81 seconds for Mr. Paterson's algorithm versus 1.51 seconds for mine. To generate an ellipse, the *plot( )* routine may be modified as Mr. Paterson suggests.

Michael P. Lindner
Basking Ridge, New Jersey

Pointers: Far vs. Huge vs. Based

Dear *DDJ*,

On page 85 in the August 1990 *DDJ*, Bruce Schatzman gives a useful description of the new *based* pointer technique available under MS C 6.0. Unfortunately, he makes some incorrect statements regarding *far* pointers.

The discussion claims that *far* pointers are the only solution available for data arrays larger than 64K. This is simply not true, as *far*, *near*, and *based* pointers all suffer from the 64K limitaion on contiguous data size. The only available solution to this limitation is the *huge* pointer.

Technically, *near*, *far*, and *based* pointers are represented using variations of segmented notation. *near* pointers assume an implied segment called DGROUP and only require a 2-byte offset. *far* pointers, on the other hand, make no assumptions about the segment. They require 4 bytes; the high word holds the segment, while the low word holds the offset. *based* pointers represent a middle ground between *near* and *far*. The segment and offset are split into two variables; one variable holds the segment, while the other holds the offset. This leads to improved efficiency where numerous references to pointers of a common segment are made.

In contrast, huge pointers are represented using 4-byte absolute notation. The use of absolute notation supports contiguous blocks of data which are larger than 64K, however, additional code overhead is incurred to transform the absolute notation into segmented notation at runtime.

Hope this untangles a few pointers and thanks for a great magazine.

Kent Funk
Manhattan, Kansas

Bruce responds: Kent is correct. Both far *pointers and* based *pointers suffer from the 64K addressing limitation, which is the size of the offset (16 bits). The reference to* far *pointers in the last paragraph of the article should have been a reference to* huge *pointers. My apologies and thanks to Kent for the correction.*

(continued on page 12)

(continued from page 8)

DOS + 386 = 4 Gigabytes

Dear *DDJ*,

In my article "DOS + 386 = 4 Gigabytes!" (*DDJ*, July 1990) there is a compatibility problem with some 386 machines. To correct this, you must replace the *a20( )* routine in Listing Three, page 110. The correct routine is:

```
/* This code is the same */
void a20(int flag)
   {
   if (inboard)    {
      outp(INBA20,flag?INBA20OFF);
      }
/* changes start here */
      else
      {
      keywait( );
      outp(0x64,0xD1);
      keywait( );
      outp(0x60,flag?0xDF:0XDD);
      keywait( );
      outp(0x64,0xFF);
      keywait( );
      }
   }
```

The code in the article works on some 386 machines, but fails on Compaq and Dell computers. The code above works on all machines I tested, including Everex, Dell, Compaq, and CompuAdd.

My apologies for any inconvenience this has caused. I'd like to thank John Hamilton of Quad-S for providing several of the test machines.

Al Williams
League City, Texas

Dear *DDJ*,

In reviewing the code supplied with "DOS + 386 = 4 Gigabytes!" by Al Williams (*DDJ*, July 1990), I have identilfied some potentially dangerous practices that are avoidable.

The first is that the state of the IRQ interrupt flag (IF) is cleared and then forcibly set at the end of the routine "protsetup." In case this routine is called while interrupts are disabled, it will reenable them. A safer method is to PUSHF, CLI, and then POPF (POPF should be avoided on code that may run on an 80286, but this is 80386 or above, only).

Generally, interrupts really are enabled in such context, so programmers tend to be lax, but it should not be left up to chance. At least the fact that interrupts will be enabled on exit should be explicitly documented.

Secondly, after changes to the protection-enabled (PE) flag, a short jump is performed to flush the prefetch queue. While this is recommended by the *80386 Programmer's Reference Manual* (Ahern-Wahlstrom, 1986) when going *into* protected mode, section 14.5 clearly states that a *far* jump is required to put "appropriate values in the access rights of the CS register [sic]."

In fact, it is desirable to do *far* jumps on both entry and exit of protected mode. This assures the kind of coherency that greatly simplifies the transition and facilitates the use of an In Circuit Emulator (ICE) or software debugger. If one is single-stepping through such code, the dumps and restores of invalid CS values could disrupt the orderly execution of instructions.

The lack of explicit CS descriptor loading should not actually cause the code to fail because no *far* transitions are performed while in protected mode. This assures the cached descriptor information will remain valid. On the other hand, this introduces an unnecessary architectural dependence that may interfere with operation on yet-to-be-released 80386 compatible processors.

I hope these recommendations help you and Mr. Williams to produce and distribute code that is as robust and widely compatible as possible.

Thomas Roden
Irvine, California

Al responds: I want to thank Thomas for his interest in "DOS + 386 = 4 Gigabytes!" I'm glad he took the time to explore it in such detail. I'd like to address his issues one at a time.

His comments about reloading the CS register for protected mode are certainly true — true, that is, if you are writing a protected-mode application. In this case, however, the same caching that allows us to use the 4-gigabyte segments allows us to use this handy technique. Of course, if the code ran with interrupts enabled, or made any long *calls or jumps, we would have to force the CS register to reload. This would have required another GDT entry, and was deemed unnecessary. Also, if you look at the code for my DOS extender (Part I in this issue), the CS register is, in fact, loaded as you suggest. Thanks again and I hope I have answered any concerns.*

[Editor's note: Thomas is the author of "Four Gigabytes in Real Mode," published in Programmer's Journal *(November/December 1989).]*

Zortech Heard From

Dear *DDJ*,

It was with considerable interest that I read the article titled "Collections in Turbo C++" by Bruce Eckel in your August 1990 issue.

The article seems to suggest that factors such as compilation speed, code size and speed, and support for development of Microsoft Windows 3.0 and OS/2 are irrelevant or certainly not worth his mention. For those interested in such features, Zortech C++ remains the platform of choice. Our benchmarks clearly show Turbo C++ compiles slower, and generates bigger, slower code. Additionally, Turbo C++ currently has no support for the Windows and OS/2 operating environments.

Surprisingly, a big deal was made of Zortech not supporting the C++ language feature *pointers to members*. As of July 1, Zortech introduced this feature in its version 2.1 release. It's particularly surprising, since at the time the article was submitted, Bruce Eckel and *DDJ* had beta copies in their possession that clearly implemented this feature.

Perhaps of greatest concern was that Bruce Eckel's business relationship with Borland International was not made clear on the front of the article in the traditional fashion. It is standard journalistic practice to reveal to readers any conflicts of interest at the time of publication.

Paul Leathers
President, Zortech Inc.

DDJ responds: First of all, the article was about collections in Turbo C++, not *collections in Zortech C++. No comparisons were made or intended so there was no reason to discuss Zortech's features, admirable as they may be. As you said, when the article was written, Zortech C++ 2.1 was still in beta.* DDJ *covers software only when readers can buy it off the shelf.*

There was no tie between Bruce and Borland when Bruce wrote the article. After the magazine was available on newsstands, Bruce began teaching an "Introduction to C++" seminar as part of Borland's roving OOP educational program. You're right-on about magazines publishing at the front of the article *any of the author's interests that might suggest possible bias. Happily,* DDJ *is one of the few magazines to adhere to this practice.*

Editor Update

Dear *DDJ*,

I have to take exception to the sidebar "Regular Expressions" in the article "Awk as a C Code Generator," by Wahhab Baldwin (August 1990). The Microsoft Editor (M) does indeed support parentheses and + in regular expressions. In fact, it supports two flavors of each.

Parentheses may be used for simple

(continued on page 14)

(continued from page 12)
grouping, as is typical. In addition, braces provide grouping and retain the matched substring for use in a replacement expression. This is done by specifying $1, $2, etc., in the replacement expression to indicate what matched the first pair of braces, the second pair, etc.

The simple plus sign is supported, and is defined to match the shortest string that gives a successful match for the regular expression as a whole. Also, the hash sign is also used to mean one or more occurances of the previous subexpression, but in this case it is defined to match the maximum number of occurences that gives a match for the expression as a whole. The difference between these two forms is only of significance in search and replace operations. For example, replacing *foo(bar)+* with *spam* in the text *foobarbar* will give *spambar*. On the other hand, replacing *foo(bar)#* with *spam* in the text *foobarbar* will give *spam*.

Paul J. Ste. Marie
CIS 72200,1324

Guess Who's Coming to Dinner?

Dear *DDJ*,

Shocking but true. In the August 1990 *DDJ*, both Douglas N. Franklin and Michael Swaine are right in their assertions about the "Dining Philosophers" problem. Mr. Franklin suggests it can be solved and Michael suggests it cannot. The key is that they are talking about different forms of the same problem. The problem solved by Mr. Franklin is not that same problem proved unsolvable by deterministic algorithms in *Algorithmics* by David Harel (Addison-Wesley, 1987). To quote the problem from Mr. Harel on page 288:

> Can the dining philosophers problem be solved without a doorman [Mr. Franklin's first solution], and without resorting to shared memory [Mr. Franklin's second solution] or its equivalents? In other words, is there a fully distributed, fully symmetric solution to the problem that does not employ any additional processors? Here "fully distributed" means that there is no central shared memory and the protocols may use only distributed variables . . . . "Fully symmetric" means that the protocols for all philosophers are essentially identical [ruling out Mr. Franklin's third solution].

Mr. Harel goes on to prove that this highly restricted form of the dining philosophers problem can't be solved with deterministic algorithms. Mr. Harel provides the following algorithm, page 304:

```
1     do the following again and
      again forever:
1.1   carry out private activities un-
      til hungry;
1.2   toss coin to choose a direc-
      tion, left or right, at random;
1.3   wait until fork lying in cho-
      sen direction is available, and
      then lift it;
1.4   if other fork is not available
      do the following:
1.4.1 put down fork that was lifted;
1.4.2 go to 1.2;
1.5   otherwise (i.e. other fork is
      available) lift other fork;
1.6   critical section: eat to your
      heart's content;
1.7   put down both forks (and
      back to 1.1);
```

and then proves that it solves the restricted form of the problem. Note that this algorithm depends on randomness, the point of Mr. Swaine using the reference in his article.

It is unfortunate Mr. Swaine did not make it clear that Mr. Harel was referring to this highly restricted form of the problem, not the classic form solved by Mr. Franklin. But, Mr. Franklin should have given Mr. Swaine the benefit of the doubt and looked up *Algorithmics* in his local library, which I consider Mr. Franklin's homework, before accusing Mr. Swaine of not doing his. I suggest Mr. Franklin, or anyone else even marginally interested in the theory of algorithms, pick up a copy of *Algorithmics*. It is an excellent book.

Charles P. Jazdzewski
Watsonville, California

Errata: Correction for Example 1, "Dining Philosophers" problem, August 1990 "Letters."

```
#define N 5
#define take_fork(num) down(s[num])
#define put_fork(num)  up(s[num])
typedef int semaphore;
semaphore s[N];
philosopher(i)
int i;
{
     while (TRUE) {
      think();
      take_fork(i);
      take_fork((i+1) % N);
      eat();
      put_fork(i);
      put_fork((i+1) % N);
     }
}
```

DDJ

We welcome your comments (and suggestions). Mail your letters (include disk if your letter is lengthy or contains code) to DDJ, *501 Galveston Dr., Redwood City, CA 94063, or send them electronically to CompuServe 76704,50 or via MCI Mail, c/o* DDJ. *Please include your name, city, and state. We reserve the right to edit letters.*

Roll Your Own
DOS Extender: Part I

Develop your own 386 protected-mode applications

Al Williams

The 80386 is chock-full of advanced features that support modern applications and make programming easier. Unfortunately, DOS programmers can't take advantage of many of these features because DOS is unable to use the 386's special protected mode. So what can you do when you need to write PC programs that require large amounts of memory, multitasking, or other sophisticated features?

One solution is to move to a protected-mode operating system such as Xenix 386 or OS/2. Another approach, however, is to turn to a DOS extender that provides some mechanism for interrupt-driven I/O and for making DOS and BIOS calls in a protected-mode program. Some DOS extenders switch between real and protected mode to handle interrupts and make DOS calls, but the preferred method runs DOS in virtual-86 mode, which causes the 80386 to emulate an 8086. (See the accompanying text box entitled "Protected Mode Operations on a PC" for more details.)

Actually, running protected-mode programs on a 386 is not very difficult if you don't need to access the system calls or perform any interrupt-driven I/O. In fact, two articles previously published in *DDJ* did just that (see the references at the end of this article). However, your programs will often need to do disk and keyboard I/O, as well as make calls to DOS and the BIOS.

In this two-part article, I present a DOS extender, PROT, that deals with interrupt-driven I/O along with most DOS and BIOS calls in protected mode using virtual-86 mode. In this first installment, I'll discuss protected mode in general and the basics of PROT in particular. In next month's issue, I'll cover debugging issues and 80386 exceptions and take you under the DOS extender's hood.

Al is a systems engineer on the space station Freedom project for Jackson and Associates. Look for an expanded version of PROT in his book DOS: A Developer's Guide, *which will be available from M&T Books early in 1991. Al can be reached at 310 Ivy Glen Ct., League City, TX 77573, or via CompuServe at 72010,3574.*

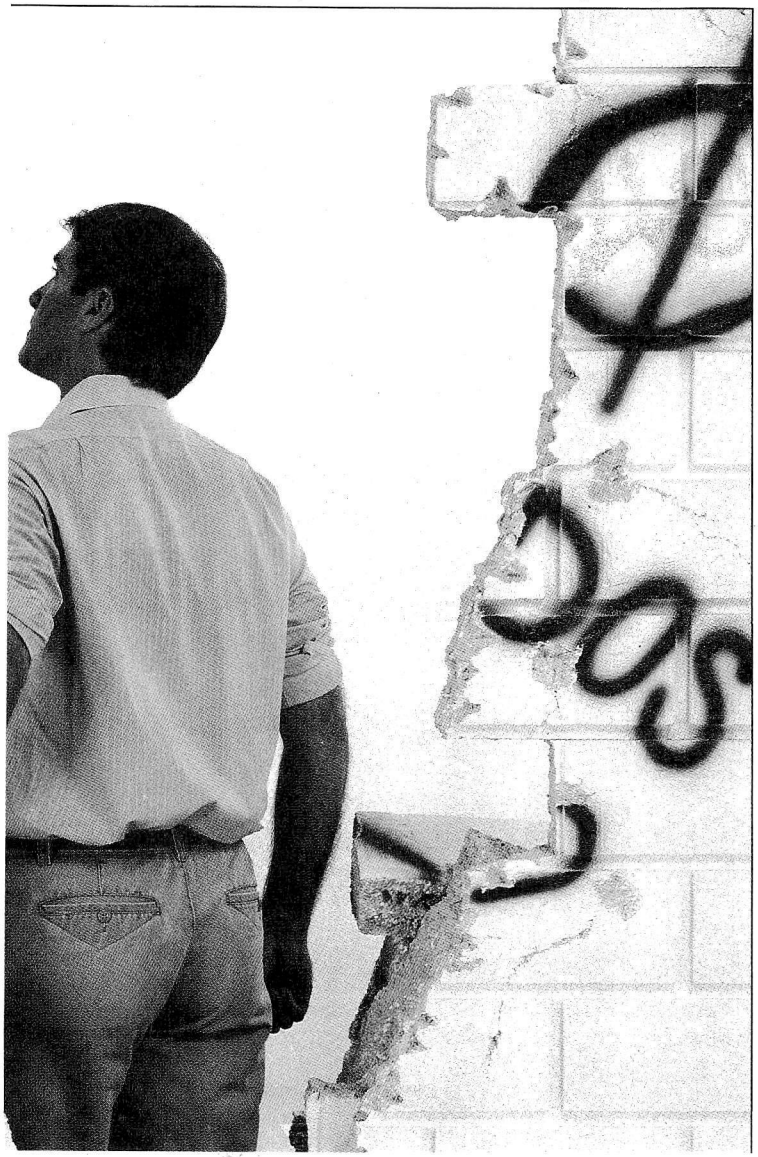


To implement this system, you'll need Microsoft's MASM 5.1 or Borland's TASM and an AT-style computer with an 80386, 80486, or 80386SX CPU, or an Intel Inboard 386/PC. While all of the source code listings (more than 2000 lines) that accompany this two-part article will be available online with this issue, I'll cover Listings One through Three in part one and Listings Four through Seven in part two.

About PROT

PROT.ASM (see Listing One, page 81) and its associated *include* files make up a true, 32-bit DOS extender. This extender allows you to write assembly language programs that use 32-bit addressing and access all of the 80386's special features. In addition, PROT allows you to do I/O using the ROM BIOS or DOS. PROT also has provisions for direct access to the PC hardware (for instance, to write directly to the screen).

PROT doesn't allow you to call interrupts that terminate your program (such as INT 20H or INT 21H function 4CH); instead, PROT provides its own calls for program termination. Obviously, the two BIOS calls that switch the processor into protected mode (INT 15H functions 87H and 89H) won't operate properly. (Their functions are superfluous under PROT anyway.) Since PROT doesn't allow program termination calls, it cannot deal with spawning DOS subprocesses using INT 21H function 4BH, nor does it allow the undocumented DOS command processor "backdoor" interrupt (INT 2EH). Of course, if you must spawn a subprocess, you can always return to real mode temporarily.



PROT does include macros to assemble some 32-bit instructions since the linker that comes with MASM doesn't handle certain 32-bit references properly. These macros are particularly useful when the assembler generates a negative 32-bit relative number. In that case, the linker only fills in the bottom 16 bits of the number, which changes the negative relative jump into a positive jump. The supplied macros overcome this difficulty.

The Segments

Any program written with PROT starts with 23 segments that are defined in the GDT (although you can define more in your program). PROT does not set up an LDT, but your code can easily set one up if you require it. The segments your programs will use are shown in Table 1.

When your program runs, the segment registers are initialized to the values shown in Table 2. GDT.INC (Listing Two, page 86) contains the names of all of the predefined segment descriptors. Your program will begin as a privilege level 0 task.

PROT's other components include EQUMAC.INC (Listing Three, page 86), discussed in detail below; STACKS.INC (Listing Four), which contains the stack segments; INT386.INC (Listing Five) for 386 interrupt handling; TSS.INC (Listing Six), which contains the task state segment definitions; and CODE16.INC (Listing Seven), which is the 16-bit DOS entry/exit code. I'll cover the last four in next month's installment.

Writing a Program

Your program goes into a file with a .PM (protected mode) extension. It should consist of the two user segments SEL_UDATA and SEL_UCODE. Execution begins with the *USER* procedure.

Example 1 shows the simplest possible PROT program; it does nothing except return to DOS. The *NODATA* macro declares an empty data segment, since the program uses no data. The line *BACK2DOS* is equivalent to *JMPABS32 SEL_CODE16,BACK16,* which returns to DOS. If you load a value in the AL register before making this jump, DOS receives that value as the return code. The *BACK2DOS* macro accepts an optional argument, which the macro loads into AL for you. PROT will also return to DOS if a breakpoint or an unexpected interrupt occurs. In this case, DOS receives a return code of 7FH.

The *PROT_CODE* and *PROT_CODE_END* statements are actually macros defined in EQUMAC.INC (Listing Three). Use these macros to define your main code segment, as shown in Example 1. The corresponding *PROT_DATA* and *PROT_DATA_END* macros allow you to define your main data segment if needed.

Most programs will make calls to DOS or the BIOS. In PROT, the *call86* routine makes this possible. This routine takes a pointer (in ES:EBX) to a parameter block (see Figure 1). A macro, *VM86CALL*, performs the far call to *call86*.

Example 2 shows a short DOS program that prints a message using DOS function 9 and the corresponding program written with PROT. The statement *PROT_STARTUP* (again, a macro in EQUMAC.INC) sets the default parameter block's data segment and stack. You can override these defaults when you call *PROT_STARTUP*.

When calling *call86*, all registers except the segment registers, EFLAGS, EBX, and EBP are passed to the VM86 interrupt unchanged. EBX, EBP, EFLAGS, and the segment registers receive their values from the parameter block. If you want the segment registers returned in the parameter block, set the first word in the block to a non-zero value. Otherwise, the parameter block remains unchanged. Upon return, all non-segment registers will contain the values

returned by the *VM86* call.

The SEL_DATA segment defines a default parameter block (*pintframe*). You may use this for all of your DOS calls, or for better performance you can define multiple blocks by using the *vm86blk* structure in EQUMAC.INC. For instance, you might define three different blocks: one for disk reads, one for BIOS screen writes, and another for other BIOS calls. Do not use the other parameter blocks defined in SEL_DATA (*hintframe* and *cintframe*) in your programs. These parameter blocks handle hardware interrupts and critical errors exclusively.

Whenever you pass addresses to DOS and BIOS routines, you must ensure that they point somewhere in the first megabyte of memory. If you are using a lot of extended memory areas for storage, it might be wise to allocate one or two temporary storage areas in low memory just to handle DOS calls.

By default, PROT ignores Ctrl-C interrupts. Your program can test the flag *breakkey* in the SEL_DATA segment to see if a break event occurred. You can set the locations *break_seg* and *break_off* to the address of your own protected-mode break handler if you wish. The routine pointed to will execute after calling a DOS or BIOS routine with *call86* if a break has occured. PROT also ignores the Ctrl-Alt-Del keystroke that normally reboots the computer, since rebooting in protected mode will cause the system to crash.

PROT provides a default critical error handler similar to the one found in DOS. By setting *crit_seg* to 0 you can

Segment	Function
SEL_DATA0	4-gigabyte data segment starting at location 0. With this segment, you can address any memory location you please. Be careful.
SEL_GDT	Alias for the GDT. You may need this to add more segments or find information about the predefined segments.
SEL_VIDEO	4K-data segment at video page 0. PROT determines your video adapter type, sets the page to 0, and sets SEL_VIDEO to the proper address.
SEL_DATA	Contains PROT's system data area. Several useful variables reside in this segment.
SEL_IDT	Alias for the protected-mode interrupt vector table. You may wish to modify this segment so you can add interrupts to the system.
SEL_UCODE	Your program's default code segment.
SEL_UDATA	Your program's default data segment.
SEL_PSP	256-byte long data segment that contains your program's DOS PSP. You can use this segment to access the command line and other MS-DOS specific data.
SEL_ENV	Contains your program's DOS environment block.
SEL_FREE	Starts at the first free location of DOS memory and goes to the end of DOS RAM (640K or less).
SEL_EFREE	Similar to SEL_FREE, but begins at the start of extended memory and continues to the end of extended memory as reported by INT 15H, function 88H. If no extended memory exists, SEL_EFREE will have a limit of 0.

Table 1: *Segments used by a PROT program*

```
DS=SEL_UDATA
ES=SEL_DATA
FS=SEL_DATA0
GS=SEL_VIDEO
```

Table 2: *Initial segment register values*

```
File: USER.INC
; SET UP EMPTY DATA SEGMENT
          NODATA

; SET UP CODE SEGMENT - PROGRAM RETURNS TO DOS
          PROT_CODE
USER      PROC NEAR
          BACK2DOS
USER      ENDP
          PROT_CODE_END
```

Example 1: *A simple PROT program*

Address		Member name
BLOCK+0	Segment register flag (see text)	VMSEGFLAG
BLOCK+4	Interrupt number	VMINT
BLOCK+8	EFLAGS	VMFLAGS
BLOCK+12	ESP	VMESP
BLOCK+16	SS	VMSS
BLOCK+20	ES	VMES
BLOCK+24	DS	VMDS
BLOCK+28	FS	VMFS
BLOCK+32	GS	VMGS
BLOCK+36	EBP	VMEBP
BLOCK+40	EBX	VMEBX

Figure 1: *Parameter block for* call86 *routine*

Real Mode Program

```
REALPGM   PROC
          MOV AX,SEG STACKAREA
          MOV SS,AX
          MOV SP,OFFSET STACKAREA    ; SET UP STACK
          MOV AX,SEG DATSEG
          MOV DS,AX                  ; SET UP DATA SEGMENT
          MOV DX,OFFSET MESSAGE      ; LOAD POINTER TO MESSAGE
          MOV AH,9
          INT 21H                    ; PRINT MESSAGE
          MOV AH,4CH
          INT 21H                    ; RETURN TO DOS
REALPGM   ENDP
```

PROT Equivalent

```
USER      PROC
          PROT_STARTUP               ; SET UP STACK/DS
          MOV AX,21H
          MOV PINTFRAME.VMINT,EAX
          MOV EDX,OFFSET MESSAGE     ; LOAD POINTER TO MESSAGE
          MOV AH,9
          MOV EBX,OFFSET PINTFRAME
          VM86CALL                   ; PRINT MESSAGE
          BACK2DOS                   ; RETURN TO DOS
USER      ENDP
```

Example 2: *DOS and PROT code fragments to print a message using DOS service 9*

(continued from page 18)
completely disable critical error handling and PROT will ignore critical errors. You can set *crit_seg* and *crit_off* to the segment and offset of your own critical error handler. A protected-mode critical error handler is very similar to a normal real-mode error handler. A real-mode handler gets status information in AX, DI, BP, and SI. For protected-mode handlers, the AX value is in *critax;* the DI, BP, and SI values are in *critdi, critbp,* and *critsi,* respectively. Your error handler must return a value in AL that determines the action to take. If AL is 0, PROT will fail the error. If it is 1, PROT retries the error. And if AL is equal to 2, PROT will abort to DOS. If you choose to abort the program due to a critical error, PROT returns a 7FH to DOS.

Two DOS interrupts, INT 25H and 26H, do not properly return to their callers. They normally leave the caller's flags on the stack when returning. When programming with PROT in protected mode, these flags do not remain on the stack. The same effect can be obtained with the code shown in Example 3. However, this is not a problem when running programs in virtual-86 mode, but only in protected mode. Of course, if you don't need the old flags, and you usually won't, you don't need to worry.

PROT uses several routines that may also be useful to the applications programmer, as shown in Table 3. Your programs can call these routines via a far call (the *CALL32F* macro).

Putting It Together

After creating a source file and setting any equates that you want to change at the top of EQUMAC.INC, you can compile a PROT application by using the batch file shown in Figure 2. The resulting EXE file will execute from the DOS prompt. If PROT does not find a 386 or 486, it will exit with an error message and return 80H to DOS. PROT will also exit with an 80H to DOS if another program already has the computer in protected mode.

(continued on page 24)

Protected-Mode Operations on a PC

If protected mode is so great, why isn't it used more often on 386 PCs? There are a host of difficulties in trying to get a DOS-based PC to do anything useful in protected mode. The primary difficulty is DOS itself; DOS expects to run in real mode. The same goes for the ROM BIOS (except in the PS/2-type computers). To remain compatible with the old 8088-based PCs, all 386 PCs have one of their address lines switched off to prevent accessing memory above 1 Mbyte. Finally, some of the hardware interrupts used by the PC conflict with the interrupts the 80386 uses for error handling in protected mode.

With all of these obstacles, the prospect of running protected-mode programs on a DOS-based PC seems bleak. However, the PC's address lines and interrupt controllers are reprogrammable. Better still, the 80386 has a special mode (virtual-86 or VM86 mode) that allows old-style 8086 programs (like DOS or the BIOS) to operate in protected mode.

According to the Intel documentation, running 8086 code in VM86 mode is fairly straightforward. However, attempting to implement Intel's strategy fails when it comes to the PC's BIOS and DOS. Intel assumes that your 8086 code will always call interrupts with an *INT* instruction and return with an *IRET* instruction. However, this seems to be the exception rather than the rule with the PC's system code. Some DOS extenders deal with this problem by returning to real mode for each system call, then switching back to protected mode upon completion. While this is fairly easy to implement, it causes problems if your programs are written to take advantage of the 80386's multitasking capabilities.

To run the BIOS and DOS in VM86 mode, you have to provide the 386 with a VM86 task. You have to emulate certain instructions, most notably interrupts. You also have to reprogram the hardware interrupt controllers and redirect their interrupts to the proper routines.

A VM86 task requires emulation for the *CLI, STI, LOCK, PUSHF, POPF, INT,* and *IRET* instructions. This is to prevent the VM86 task from disrupting other tasks that might be running under protected mode. PROT emulates all of these instructions except *LOCK,* which isn't really an instruction but a prefix. Only multiprocessor systems use *LOCK,* so PC software runs fine without it.

In theory, emulating *INT* and *IRET* is fairly straightforward. The execution of an *INT* or *IRET* in VM86 mode causes a general protection exception (*INT 13*). When you detect an *INT* instruction, simply determine the required interrupt vector address, simulate the interrupt, catch the corresponding *IRET,* and return to the calling program. In practice, the PC BIOS and DOS do not always have a one-to-one correspondence between *INT*s and *IRET*s (a problem we will explore in detail next month). Only the normal *INT/IRET* sequence provides the *INT 13* required to emulate these instructions.

Some DOS extenders take different approaches. Some actually switch the processor back to real mode for each call to DOS or the BIOS. Other VM86 programs (such as EMS memory simulators) let real-mode calls run unprotected, which shuts you off from many of the 80386's special features and only allows DOS calls from VM86 mode. PROT actually runs DOS and the BIOS as a VM86 task.

Note that some of the protected-mode features that are also available in real mode are unavailable in VM86 mode. For example, a VM86 task can't switch the processor into protected mode in the same way a real-mode program can. This means some 386-specific software may not run with PROT. Also, some very specific BIOS routines that deal with extended memory and protected mode may not work. However, with protected-mode programming, you won't need BIOS services to manage extended memory or switch modes.

PROT provides facilities to handle Control-C interrupts and critical device errors in protected mode. By default, PROT ignores Control-C interrupts and has a critical error handler similar to the one provided by DOS. PROT also catches and ignores the Ctrl-Alt-Del keystroke that normally resets the computer since the PC's BIOS won't reboot in protected mode.

PROT reprograms the interrupt controllers so that hardware interrupts can coexist with 80386 exceptions. When PROT detects a hardware interrupt, PROT automatically redirects it to the proper BIOS or DOS interrupt handler.

—A.W.

(continued from page 20)

I've included two short sample programs to illustrate PROT programming and a protected-mode file browser, FBROWSE.PM. Due to space limitations, however, these programs are not included in the listings section, but are available directly through *DDJ*.

That's it for this issue. Next month I'll dive into debugging, 386 exceptions, and take a more in-depth look at PROT itself.

Bibliography

Green, Thomas, "80386 Protected Mode and Multitasking," *Dr. Dobb's Journal*, September 1989: pp. 64–72.

Intel Corporation, *80386 Programmer's Reference Manual*, Santa Clara, Calif.: Intel Corporation, 1986.

Margulis, Neil, "Advanced 80386 Memory Management," *Dr. Dobb's Journal*, April 1989: pp. 24–30.

Margulis, Neil, "80386 Protected Mode Initialization," *Dr. Dobb's Journal*, October 1988: pp. 36–39.

Turley, James L. *Advanced 80386 Programming Techniques*, Berkeley, Calif.: Osborne/McGraw-Hill, 1988.

Williams, Al, "Homegrown Debugging — 386 Style!" *Dr. Dobb's Journal*, March 1990: pp. 46–57.

```
MOV EAX,25H
MOV PINTFRAME.VMINT,EAX
PUSHF                       ; (Or PUSHFD)
VM86CALL                    ; Call INT 25 or 26
        .
        .
```

Example 3: *Maintaining the caller's flags on the stack when returning in protected mode*

Routine	Purpose
CLS	Clears page 0 of the video display directly.
OUCH	Prints the character in AL to page 0 of the video display using direct video access.
CRLF	Perform a carriage return/line feed using the OUCH routine.
MESSOUT	Prints the zero-terminated string pointed to by DS:EDX using OUCH. Modifies EBX.
HEXOUT	Outputs the byte in AL in hex using OUCH.
HEXOUT2	Outputs the word in AX in hex using OUCH.
HEXOUT4	Outputs the double word in EAX in hex using OUCH.
MAKE_GATE	Makes a task gate, trap gate, interrupt gate, or call gate. Call this routine with ES:EDX pointing to the table's (GDT, LDT, or IDT) base address (as a read/write segment). Set CX to the target descriptor, EBX to the target offset (if applicable), SI to the selector for the gate, AH to one of the access right bytes (ARB) defined in EQUMAC.INC (Listing Two), and AL to the word count (for call gates only).
MAKE_SEG	Makes a segment descriptor. Call this routine with ES:EDX pointing to the GDT or LDT table base address (as a read/write data segment). EBX is the base address of the segment, ECX is the limit (in bytes), AL is 0 for a 16-bit segment or 1 for a 32-bit segment, and AH is one of the access rights bytes (ARB) defined in EQUMAC.INC.

Table 3: *PROT's programmer callable routines*

```
echo off
if X%1==X goto :errexit
if NOT X%2==X goto :errexit
masm /DPROGRAM=%1 PROT.ASM,%1.OBJ,%1.LST;
if ERRORLEVEL 1 goto :exit
link %1;
goto :exit
:errexit
echo PMASM - An MASM driver for the PROT 386 DOS Extender
echo usage: PMASM progname
echo  Assembles the file progname.pm into progname.exe
echo  The PROT system is copyright (C), 1989 by Al
echo  Williams.
echo  Please see the file "PROT.ASM" for more details.
:exit
```

Figure 2: *Batch file used to compile a PROT program with MASM*

Source Code Availability

As a service to our readers, all source code is available on a single disk and on line. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Drive, Redwood City, CA 94063, or call 800-356-2002 (inside Calif.) or 800-533-4372 (outside Calif.). Specify issue number and disk format. Code is also available through M&T's Telepath online service (via TYMNET) and through the DDJ Forum on CompuServe (type GO DDJ).

DDJ

(Listings begin on page 81.)

Vote for your favorite feature/article.
Circle Reader Service **No. 1.**

Opening OS/2's Backdoor

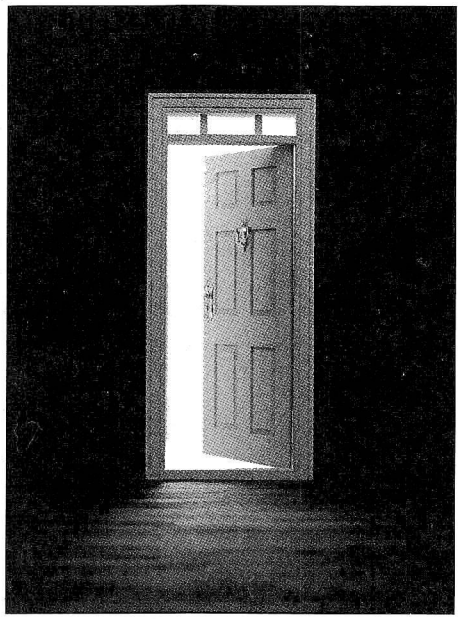
DEVHLP.SYS provides low-level services

Andrew Schulman

Unlike real-mode MS-DOS, which gives the programmer total access to its 1-Mbyte sandbox, the protected-mode OS/2 operating system takes strict control over its vastly larger address space (up to 16 Mbytes of physical memory mapped into up to 1 gigabyte of virtual memory). Strict control over memory is necessary not only because there is so much more of it in OS/2, but also because OS/2 supports multitasking (again in contrast to MS-DOS, which only *allows* multitasking via interrupt handlers).

However, "there ain't no such thing as a free lunch." OS/2's tight management of memory means that an application can't freely peek and poke arbitrary locations in memory. While this is good because it prevents tasks from bashing the operating system (or each other), it does make it nearly impossible for applications to perform memory-mapped I/O or access memory on adapter cards.

It also makes it difficult to write OS/2 diagnostic tools. For example, probably every PC programmer has a utility such as CORELOOK, MAPMEM, RAMSCAN, or XRAY in his or her \BIN directory. How would you port such a



utility to OS/2? Or how would you write a program that displays all named semaphores in the system? (Hint: Walk through memory, looking for strings that begin with the pattern "\SEM\". The real puzzle, of course, is how to walk through all of memory.)

The OS/2 application program interface (API) does have one function that seems to help with this task: *VioGetPhysBuf( )*. You pass in a 32-bit absolute physical address, presumably corresponding to a video adapter's display buffer, and *VioGetPhysBuf( )* returns a selector that can be used to manipulate the buffer directly. There's nothing to prevent us from using this to manipulate memory *not* on a video adapter.

The VIOPHYSBUF structure that *VioGetPhysBuf( )* takes has one crucial restriction, however: The physical address must be in the range *A0000h* through *BFFFFh*. This makes some sense because, without restrictions, the single *VioGetPhysBuf( )* entry point could be used as a Trojan horse to defeat OS/2 memory protection. But then we're stuck with the riddle of how to examine physical memory locations outside the range *A0000-BFFFF*.

With mounting irritation, we comb through IBM's OS/2 documentation looking for some function that will do the trick (in API-rich environments such as OS/2 or Windows, programming seems to have been reduced to the ability to search through a parts catalog). Finally, in the section on device drivers, we find something that sounds like what we're looking for:

PhysToUVirt
Map Physical To User Virtual Address

IBM's documentation says that this "converts a 32-bit physical address to a valid selector offset pair addressable out of the current LDT." This is, in fact, exactly what's needed to do memory-mapped I/O: Bang on adapter cards and walk through physical memory.

PhysToUVirt is a *DevHlp*, one of about 55 helper routines OS/2 provides for device drivers. In his indispensable book

Andrew is an engineer/writer at Phar Lap Software in Cambridge, Mass. He is a contributing editor for Dr. Dobb's Journal. *Andrew can be reached by phone at 617-876-2102 or on CompuServe at 76320,302.*

(continued from page 28)
Inside OS/2 (Microsoft Press, 1988), Gordon Letwin refers to the *DevHlps* as OS/2's "backdoor." Many of the device helpers have capabilities found nowhere else in OS/2. A few examples: *ABIOSCommonEntry* and *ABIOSCall* (call the PS/2 ROM BIOS), *AllocPhys* (allocates physical memory), *Lock* (lock virtual memory against relocation or swap), *RealToProt* (switch from real to protected mode), *SetIRQ* (attach an interrupt handler to an IRQ level), and *VirtToPhys* (convert locked virtual address to physical address).

There's one hitch: Only device drivers can call the *DevHlps*. Unlike the rest of the OS/2 API, device helpers use a register-based, parameter-passing mechanism much like the "old" (but by no means defunct) MS-DOS *INT 21* interface. All device helpers share a common entry point, whose address is made known to the driver only at initialization time. Individual functions (such as *PhysToUVirt*) are specified with a number in the DL register.

Figure 1 shows a sample invocation of the *PhysToUVirt* device helper. A moment's consideration should convince you that this facility to map physical addresses into a user's address space is all that is required to circumvent any inconveniences of protected mode. Combined with the Intel SGDT instruction, the *PhysToUVirt DevHlp* could be used to control the protected-mode environment. OS/2's designers probably made a wise choice in allowing this function to be called only from a device driver.

Now, you could get the company's device-driver guru to write a separate OS/2 device driver to accompany each application that you wish could call a *DevHlp*. For example, to use *PhysToUVirt*, write a device driver that maps access to physical memory onto the read and write operations. This would resemble the Unix */dev/mem* device: A memory fetch is a read, and a store is a write; memory is just another stream of bytes.

This is a good idea, but the resulting driver only supports the *PhysToUVirt DevHlp*. When you realize that you also need the *VirtToPhys DevHlp*, you're out of luck. Isn't there any way for a "normal" OS/2 Ring 3 application to call any device helper?

There is. Using the function *DosDev-*

Walking the OS/2 Device Chain

Arthur Rothstein

DRVRLIST.EXE is a DEVHLP application that lists the device drivers in OS/2. The design considerations are similar to those of the analogous DOS program. Descriptive information about a driver is contained in its device control block (DCB), and the DCBs are linked via a double-word pointer at offset 0. The last DCB in the chain has a pointer with offset *0FFFFH*. The challenge is to find the first in the chain.

The first two DCBs are NUL and CON, and both are in OS/2's global data segment. This segment is mapped by a bimodal selector, one that describes the same physical memory whether the CPU is in real or protected mode. The bimodal selector varies with the maintenance level of the kernel. The segment is also mapped by selector 50H, which does not vary with the kernel maintenance level (it even works with IBM SE 1.2). The program, source code for which appears in DRVRLIST.C in Listing Two (page 96), and which uses the header file DEVHLP.H in Listing Three (page 97), searches for the first occurrence of the driver name, "NUL" followed by five blanks, then backs up 10 bytes. When displaying the driver address, the program uses the bimodal selector instead of 50H. Here is sample output from DRVRLIST under OS/2 1.1:

```
Address Name
9C0:03EE NUL
9C0:040A CON
798:0000 COM1
748:0000 SINGLEQ$
730:0000 MOUSE$
720:0000 POINTER$
6D0:0000 DEVHLPXX
390:0000 Block device, 3 logical units
380:0000 PRN
380:001A LPT1
380:0034 LPT2
380:004E LPT3
360:0000 KBD$
350:0000 SCREEN$
340:0000 CLOCK$
```

The DCBs are mapped by GDT selectors, which the program, running at privilege Level 3, cannot access. Given a GDT selector, the program uses the *VirtToPhys* device helper to get the physical address mapped by the selector, then the *PhysToUVirt* device helper to get an LDT selector to the same physical memory. These two calls in sequence constitute function *MakeSel( )*. The LDT selector acquired by *MakeSel( )* maps a full 64K byte starting at the physical address, usually more than the GDT selector maps. The program uses this LDT selector to access a DCB. After the program finishes using a given selector, it again calls the *PhysToUVirt* device helper, this time to release the selector. The release is performed in function *ReleaseSel( )*.

The global data segment and the DCBs are fixed in memory. Their GDT selectors and the physical addresses to which these selectors map remain unchanged at least until the system is rebooted. In general the reverse is true. Selectors come and go, and those that are unchanged may map to various physical addresses during the life of the system. An example of the first is the process control block, which is assigned to a unique GDT selector during the life of a process. An example of the second is selector 28H, which maps the LDT of the active thread. When the selectors — or their physical memory — are not fixed, an application must lock the memory with the *Lock* device helper, before calling *VirtToPhys* or *PhysToUVirt* to ensure valid results. Even then you must exercise caution. We don't know, for example, how OS/2 reacts when it attempts to kill a process whose process control block has been locked by another application.

Arthur Rothstein is a programmer at Morgan Labs, a developer of PC-based transaction processing systems in San Francisco, Calif. Arthur can be reached through Morgan Labs, 690 Market St., San Francisco, CA 94104.

Dr. Dobb's JOURNAL
SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

PUBLISHER *Peter Hutchinson*

EDITORIAL
EDITOR-IN-CHIEF *Jonathan Erickson*
MANAGING EDITOR *Monica E. Berg*
TECHNICAL EDITORS *Michael Floyd, Ray Valdes*
ASSOCIATE MANAGING EDITOR *Janna Custer*
PRODUCTION EDITOR *Tami Zemel*
CONTRIBUTING EDITORS *Al Stevens, Jeff Duntemann, David Betz, Tom Genereaux, Andrew Schulman*
COPY EDITORS *Pamela Dillehay, Nan Fornal*
EDITOR-AT-LARGE *Michael Swaine*

ART/PRODUCTION
ART/PRODUCTION DIRECTOR *Larry L. Clay*
ART DIRECTOR *Michael Hollister*
PRODUCTION SUPERVISOR *Amy Shulman Lesovoy*
TYPOGRAPHERS *Charlene Carpentier, Margaret Anderson*
COVER PHOTOGRAPHER Michael Carr

CIRCULATION
DIRECTOR OF CIRCULATION *Maureen Kaminski*
CIRCULATION MANAGER *Randy Robertson*
CIRCULATION PLANNING MANAGER *Manny Sawit*
DIRECT MARKETING MANAGER *Francesca Davies*
SINGLE COPY SALES DIRECTOR *Sarah Forsman*
FULFILLMENT MANAGER *Anne Jean*
DIRECT MARKETING COORDINATOR *Susan Bauman*
PROMOTION COORDINATOR *Philip Tsang*

ADMINISTRATION
VICE PRESIDENT OF FINANCE *Kate Deschamps*
CONTROLLER *Mary Collopy*
CREDIT MANAGER *Betty Arsene*
ACCOUNTING SUPERVISOR *Renate Kernke*
ACCOUNTS RECEIVABLE *Wendy Ho*
ACCOUNTS PAYABLE *LuAnn Rocklewitz*

MARKETING/ADVERTISING
ASSOCIATE PUBLISHER *Karla Spormann*
ADVERTISING COORDINATOR *Laura Stack Pullen*
MARKETING ASSISTANT *Sara Noah Ruddy*
ACCOUNT MANAGERS *see page 160*

M&T PUBLISHING INC.
CHAIRMAN OF THE BOARD *Otmar Weber*
DIRECTOR *C. F. von Quadt*
PRESIDENT *Laird Foshay*
VICE PRESIDENT OF PUBLISHING *William P. Howard*

DR. DOBB'S JOURNAL (USPS 307690) is published monthly, except semimonthly in December, by M&T Publishing, Inc., 501 Galveston Dr., Redwood City, CA 94063; 415-366-3600. Second-class postage paid at Redwood City and at additional entry points.

ARTICLE SUBMISSIONS: Send manuscripts and disk (with article, listings, and letter to the editor) to the associate managing editor 415-366-3600.

***DDJ* ON COMPUSERVE:** Type GO DDJ.

SUBSCRIPTION: $29.97 for 1 year; $56.97 for 2 years. Foreign orders must be prepaid, including the additional postage (air or surface) in U.S. funds drawn on a U.S. bank. Add $13 per year for surface mail; add $36 for airmail to Canada and Mexico; or $26 for airlift to all other countries.

POSTMASTER: Send address changes to *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. **ISSN 1044-789X**

CUSTOMER SERVICE: For subscription questions, call toll-free 800-456-1215 (U.S. and Canada). For subscription orders or change of address call 303-447-9330 (all other countries) or write *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. For book/software orders call 800-533-4372 (in California 800-356-2002).

FOREIGN NEWSSTAND DISTRIBUTOR: Worldwide Media Service Inc., 115 E. 23rd St., New York, New York 10010; 212-420-0588 FAX 212-420-1265.

Entire contents copyright ©1990 by M&T Publishing, Inc., unless otherwise noted on specific articles. All rights reserved.

The Audit Bureau

(continued from page 30)

IOCtl( ) (equivalent to *ioctl( )* in Unix and *INT 21h AH = 44h* in MS-DOS), any OS/2 application can call a device driver, and the device driver can then call the device helper on the application's behalf. Because *DosDevIOCtl( )* allows a program to issue any device-specific commands that are not supported by other OS/2 API functions, now all we need is a device driver that acts as a *DevHlp* "server," accepting *DevHlp* requests sent to it via *ioctl* packets.

OS/2's tight management of memory means that an application can't freely peek and poke arbitrary locations in memory

I have written such a driver, called DEVHLP.SYS. It is a totally generic device driver, whose sole purpose is to call device helpers on behalf of "normal" Ring 3 applications. In one sense, it is a "fake" device driver that solely provides an *ioctl* interface without actually controlling a device; it doesn't even support the read and write functions. In another sense, though, you could say that DEVHLP.SYS provides access to the DEVHLP device. The two hundred lines of assembler source (DEVHLP.ASM) appear in Listing One (page 94).

In addition to being used for controlling devices without having to write an assembly language device driver (for example, DEVHLP.SYS is being used to interface with an image scanner from a Modula-2 program), DEVHLP.SYS has been used in several OS/2 diagnostic utilities. In the accompanying text box ("Walking the OS/2 Device Chain"), Art Rothstein describes his program DRVRLIST. In a future *DDJ* article, I will use DEVHLP.SYS to write a program for browsing the OS/2 global descriptor table (GDT).

DEVHLP.SYS works in OS/2, Versions 1.0, 1.1, and 1.2 (both Microsoft's standard and IBM's extended editions). It will definitely have to be changed for 32-bit OS/2 2.0. Note, however, that we have here a general principle: In any operating system in which device drivers are allowed operations to which regular applications are denied access, just map the device driver operations onto application *ioctl* requests to the device driver.

How to Use DEVHLP.SYS

Because DEVHLP.SYS is a device driver, it must be loaded at system boot time by adding the following line to your OS/2 CONFIG.SYS: *device = devhlp.sys*.

An application calls DEVHLP.SYS using the OS/2 function *DosDevIOCtl( )*, whose parameters are shown in Figure 2. Note that *DosDevIOCtl( )* requires a handle to a device; an application can get this handle by opening the file named "DEVHLPXX," using either the OS/2 function *DosOpen( )* or a function such as *open( )* in C.

Applications ask for *ioctl* services using a category number and a function code; DEVHLP.SYS currently provides only one service: The DEVHLP service, whose category number is 128 (the first available user-defined category) and whose function code has been more-or-less arbitrarily set to *60h*.

The two remaining *DosDevIOCtl( )*

```
MOV AX, address_high          ; top of 32-bit physical absolute address
MOV BX, address_low           ; bottom of 32-bit physical absolute address
MOV CX, length                ; count of bytes to map (0=64k)
MOV DH, request_type          ; 0=code, 1=data, 2=cancel
MOV DL, DevHlp_PhysToUVirt    ; 17h
CALL DWORD PTR [DevHlp]
JNC ok                        ; carry set=error, clear=ok
; AX contains error code
ok: ; ES:BX contains virtual address
```

Figure 1: *Invoking the* PhysToUVirt DevHlp

```
USHORT DosDevIOCtl(pvData, pvParms, usFunction, usCategory, hDevice)
PVOID pvData;          /* far pointer to data packet —>driver */
PVOID pvParms;         /* far pointer to parameter packet <—driver */
USHORT usFunction;     /* two-byte device function */
USHORT usCategory;     /* two-byte device category */
HFILE hDevice;         /* two-byte device handle */
```

Figure 2: *Parameters for calling* DosDevIOCtl( )

(continued from page 32)
parameters point to the actual request packet passed to the driver, and to a data packet into which the driver should place return values. The two pointers can be identical.

In the case of DEVHLP.SYS (because we simply want applications to be able to make *DevHlp* calls, and because the *DevHlp*s expect their parameters in the CPU registers) the request packet expected by DEVHLP.SYS is simply an image of the registers. The values an application puts in the REGS data structure should correspond exactly to what an OS/2 device driver would put in the real CPU registers. DEVHLP.SYS will load the real registers from these fields of the parameter packet, and will call *[DevHlp]*. Upon return from *DevHlp*, DEVHLP.SYS will store the contents of the registers back into the data packet.

```
typedef struct {
        USHORT ax, bx, cx, dx, si, di, ds, es, flags;
        } REGS;
```

Figure 3: *The DEVHLP parameter/data packet*

This is basically a form of "remote procedure call."

The C REGS structure shown in Figure 3, used both for the request packet passed to the driver and for the data packet passed back by the driver, closely resembles the union REGS used in the function *int86( )* provided by most MS-DOS C compilers. Of course, the same structure can be created in any programming language for which there is an OS/2 version.

DEVHLP.SYS was designed to provide the absolute minimum necessary for application access to DevHlps

Now it is simple to write a version of a *DevHlp* to be called by a Ring 3 application. For example, Figure 4 shows a C version of *PhysToUVirt( )*, which uses the OS/2-supplied macros HIUSHORT( ), LOUSHORT( ), MAKEUSHORT( ), and MAKEP( ). This version opens and closes the DEVHLPXX handle inside the function; if you expect to make a lot of calls to *PhysToUVirt( )* you could, of course, open DEVHLPXX during program initialization. Figure 4 also shows a C enumeration for the three requests handled by *PhysToUVirt*: Creating read-only executable addresses, creating read/write addresses, and releasing addresses.

Calling *PhysToUVirt( )* is also simple. For example, to examine 100 bytes at absolute location *FE008h*, refer to Figure 5(a) and Figure 5(b) to cancel the selector.

To walk through all of physical memory, just call *PhysToUVirt( )* in a loop that increments the physical address each time by 64K. To map an entire 64K into your address space at one time, pass *PhysToUVirt( )* a size parameter of zero. Break out of the loop if *PhysToUVirt( )* fails. And remember to cancel your selectors each time through the loop: Selectors are a precious resource in OS/2!

Device-Driver Documentation

In order to use DEVHLP.SYS effectively, you must know about the *DevHlp*s. The best documentation I have seen is

(continued from page 34)
IBM's *OS/2 Technical Reference 1.1: I/O Subsystems and Device Drivers*. Volume 1 contains thorough descriptions of device driver functions, *DevHlps*, and numerous *DosDevIOCtl* calls. Volume 2 describes Presentation Manager (PM) "presentation drivers."

Ray Duncan's *Advanced OS/2 Programming* (Microsoft Press, 1989) contains an excellent 70-page chapter on device drivers, with an additional 40 pages of reference material on the *DevHlps*.

Raymond Westwater's *Writing OS/2 Device Drivers* (Addison-Wesley, 1989) contains a great deal of useful information (Ray gives a course in OS/2 device drivers for "Microsoft University") but I wish it were better organized. The author sells a useful $50 toolkit for writing OS/2 device drivers in C (contact FutureWare, 842 State Rd., Princeton, NJ 08540, 201-343-2033). Unfortunately, the material on writing OS/2 device drivers in C is not included in Ray's book. Also, the book is already a little out of date, missing such *DevHlps* as *ABIOSCommonEntry*, *ABIOSCall*, and *GetLIDEntry*, all important for communicating with the OS/2-compatible Micro Channel ABIOS on IBM PS/2 machines.

Speaking of ABIOS, the key documentation for this crucial topic is Phoenix Technologies' *ABIOS for IBM PS/2 Computers and Compatibles: The Complete Guide to ROM-Based System Software for OS/2* (Addison-Wesley, 1989). The Phoenix book documents calling the *ABIOSCommonEntry* and *ABIOSCall DevHlps* from an OS/2 device driver.

Finally, as this article was going to press, IBM has released a program somewhat like DEVHLP, which reveals just about everything there is to know about OS/2 1.x memory utilization. The program is available in the PCMagnet forum on CompuServe as THESEU.ZIP (no source code is available, however).

Debugging

OS/2 device drivers can be a pain to debug. One of the reasons I wrote the generic device driver DEVHLP.SYS was to avoid having to debug any device driver other than DEVHLP.SYS itself. However, if you are writing OS/2 device drivers, in addition to the debugger that comes with Ray Westwater's package mentioned earlier, there is also the Universal Device Drive Debugger for

```
#define DevHlp_PhysToUVirt        0x17

typedef enum {
        UVirt_Exec=0, UVirt_ReadWrite, UVirt_Release
        } UVIRT_TYPE;

// turn physical address into virtual address
void far *PhysToUVirt(ULONG addr, USHORT size, UVIRT_TYPE type)
{
        REGS r;
        USHORT sel, ret=1;
        HFILE devhlp;
        r.ax = HIUSHORT(addr);
        r.bx = LOUSHORT(addr);
        r.cx = size;
        r.si = r.di = r.ds = r.es = 0; // not used
        r.dx = MAKEUSHORT(DevHlp_PhysToUVirt, type);
        if ((devhlp = open("DEVHLPXX", 0)) != -1)
        {
                ret = DosDevIOCtl(&r, &r, 0x60, 128, devhlp);
                close(devhlp);
        }
        //if DosDevIOCtl failed OR if DevHlp set carry flag . . .
        if (ret || (r.flags & 1))
                return NULL;
        else
                return MAKEP(r.es, r.bx);
}
```

Figure 4: PhysToUVirt( ) *in C*

```
(a) char far *copyright;
    copyright = PhysToUVirt(0xFE008L, 100, UVirt_ReadWrite);

(b)     #define Cancel(pv) \
            PhysToUVirt((ULONG) pv, 0, UVirt_Release)
        Cancel(copyright);
```

Figure 5: *Calling* PhysToUVirt( )

(continued from page 36)
OS/2, ($249 from OS TECHnologies, 532 Longley Rd., Groton, MA 01450, 508-448-9653).

One important feature of this debugger is its ability to operate in real, as well as protected mode. Recall that OS/2 device drivers can be *bimodal.* Few debuggers can handle bimodal applications (try using Microsoft CodeView on a DOS program that switches from real into protected mode using DPMI, for example), much less bimodal device drivers. The OSTECHnologies debugger works in the real mode as well as the protected-mode portion of an OS/2 device driver.

This brings up an important point: Device drivers are the only place in OS/2 where you can switch between real and protected mode! Microsoft and IBM seem to have designed OS/2 with the assumption that real mode was simply going to disappear by imperial edict. Why did they think that?

Design Principles

DEVHLP.SYS was designed to provide the absolute minimum necessary for application access to *DevHlp*s. No seat belts are provided! Using DEVHLP.SYS, it is trivial to crash OS/2.

There is another aspect to the driver's minimal design: Ironically, DEVHLP.SYS knows practically nothing about the *DevHlp*s! The only *DevHlp* it specifically knows about is *VerifyAccess*, which it uses to ensure that the addresses for your parameter and data packets are valid, and that the DS and ES register values you pass in are legal. Other than that, DEVHLP.SYS just blindly loads registers from the parameter packet, invokes *[DevHlp]*, and stores the registers into the data packet.

This "ignorant" design is greatly preferable to a seemingly more intelligent design that would provide separate *PhysToUVirt*, *Lock*, *ABIOSCommonEntry*, and so on, *ioctl* functions. The problem with such an approach is not only that it makes the driver larger, but also that it fails to support future *DevHlp*s. The "ignorant" design of DEVHLP.SYS is precisely what should enable it to work even with new *DevHlp*s. Such extensibility is usually given the glorified title "object-oriented programming" (see any discussion of virtual functions in C++) but really it's just common sense that the more ignorant a tool is of any specific protocol, the more likely it is to work with future protocols!

DDJ

(Listings begin on page 94.)

Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

Closing DOS's Backdoor

Gaining access to DOS without going through INT21

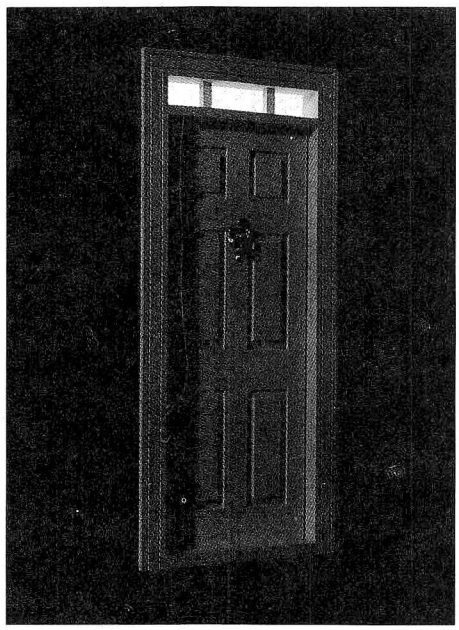
John Switzer

Although totally protecting any IBM PC or compatible from intrusion is impossible, MS-DOS complicates matters by having two "backdoors" that allow access to the DOS function handler INT 21h. These backdoors are poorly documented but still present a huge gap in a PC's security. To have a secure system, it is essential to close and lock these backdoors.

These backdoors allow access to INT 21h through two far pointers that are easily accessible by any program. The first of these pointers is in low memory, at the address reserved for interrupts 30h and 31h (0:00C0 through 0:00C7). Normally an entry in the interrupt vector table contains a *dword* pointer to the interrupt's handler; however, INT30 and INT31 are in the form of a JMP FAR instruction that points to the alternative DOS function dispatcher (in my version of DOS 3.30: JMP FAR 0274:1446). This allows direct access to DOS without having to go through INT21.

This alternative DOS handler, however, has different entry requirements than a normal INT21 call. Its use requires some special handling and an understanding of the functions that it allows. Example 1 shows the alternative entry point as it exists in MS-DOS 3.30, with some changes for clarity.

John is a technical writer for Communications Machinery Corporation and can be reached at 340 Mathilda Dr., #3, Goleta, CA 93117.



First, the handler expects the return address to be on the stack in an unusual order. Normally, when an interrupt occurs, the CPU pushes the flags onto the stack first, followed by the segment and offset of the caller's return address. However, this entry point apparently expects the flags to be pushed last, after the offset and segment of the return address. Because this routine eventually transfers control to the normal INT21h handler, the handler's first job is to translate the stack into an acceptable form for the eventual IRET. Second, this handler allows only functions 0 through 24h to be executed. Also, since the AX register is destroyed immediately upon entry, the function number is passed through CL and not through AH. Function 0Ch (CLEAR KEYBOARD BUFFER AND GET STDIN) is thus unavailable, as it uses AL for a subfunction value. These limitations may be familiar to former CP/M programmers — they result from the original MS-DOS designers' desire for CP/M compatibility.

To use this call, therefore, the caller must manually set up the stack with the flags and a proper far return address. With the function number in CL, a far jump to 0:00C0 executes the call. After completion, the INT21 dispatcher then does an IRET to the return address on the stack as normal. Example 2 demonstrates this technique.

CP/M programmers, however, used a near CALL to execute their DOS function calls. The second backdoor into MS-DOS exists precisely to duplicate this procedure: At offset 5 in every program's PSP (program segment prefix) is a far call instruction that theoretically allows access to DOS by doing a CALL 0005, as CP/M allowed. However, this offset usually shows an instruction similar to: CALL FAR F5C2:A496.

This appears to reference a location in what seems to be either the BIOS or an impossibly high RAM memory area, and the code at this address is usually garbage. So, though this pointer in the PSP has been documented since the beginnings of MS-DOS, most programmers have ignored it.

The problem is that the address shown in the PSP is usually not accu-

(continued from page 42)
rate and before use should be rounded up to the nearest paragraph. Using the previous example, this results in: CALL FAR F5C2:A4A0.

Looking at the code at this address reveals the same instruction seen at the INT30 vector (JMP FAR 0274:1446). Both backdoors, therefore, jump to the alternate DOS function dispatcher. By setting up the stack and registers as described for the first backdoor, the corrected *dword* pointer can be inserted into the program's data area. A JMP FAR instruction can then be used to execute a limited number of DOS functions.

Closing MS-DOS's backdoors removes one of the more obscure dangers to your computer and its data

By modifying the pointer in the PSP, however, the simpler CP/M approach could also be used to access DOS functions. First, the PSP is modified (if necessary) to round the pointer up to the nearest paragraph. The caller can then use a CALL 0005 instruction to execute the DOS function, as it was done in CP/M. This pushes the caller's correct return address onto the stack and executes the far CALL in the PSP. The far CALL pushes the program's code segment onto the stack, as well as another return address; however, the second return address is pointing to offset 0Ah in the PSP. This is not a problem, because the first thing the DOS dispatcher does is eliminate the second return address with a POP AX instruction (see Example 1). It then rebuilds the stack so that the IRET correctly returns to the caller's program. So, with some work, the original designer's goal of CP/M compatibility is achieved, for whatever it is worth.

Now that the alternative DOS handler is understood, its use must be prevented. Most security programs do an admirable job of closing the door to normal DOS calls using INT 21h, but many ignore these alternative entry points into DOS. A secure system must close these backdoors. For the backdoor at 0:00C0, this is trivial. Simply replace the JMP FAR instruction at 0:00C0 with a pointer to a new handler

(continued on page 47)

(continued from page 44)
that can refuse or execute the function as appropriate.

The second backdoor, however, seems to be more difficult. Because any number of PSPs can exist in memory at one time, patching each one with a new vector could be difficult. Fortunately, this is not necessary. Doing some calculations on the modified address given in the PSP (F5C2:A4A0, for example) shows that it translates into 0:00C0 because of the quirks of the "wrap-around" memory addressing found in the real-mode of the Intel IBM processors: Offset A4A0 = segment 0A4A, so segment F5C2 plus segment A4A = segment 1000C. Segment 1000C wraps around to segment 000C which translates into address 0:00C0. Thus, changing the PSP pointers is unnecessary, since both backdoors are different pointers to the same memory location. Changing the vector at 0:00C0 adequately protects against both.

Both of these backdoors have existed in PC-DOS since version 1.0 and in most versions of MS-DOS. Given that they have existed for almost nine years without causing any apparent prob-

```
ALT_DOS_ENTRY:
        POP AX                  ; get rid of flags
        POP AX                  ; save caller's segment
        POP CS:TEMP             ; save caller's offset
        PUSHF                   ; save flags
        CLI                     ; kill interrupts
        PUSH AX                 ; save caller's segment
        PUSH CS:TEMP            ; save caller's offset
        CMP CL,24h              ; is CL < max #?
        JA REFUSE_RQST          ; no, so invalid
        MOV AH,CL               ; yes, AH=function #
        JMP CONT_INT21          ; and continue INT21
```

Example 1: *An alternative entry point into MS-DOS 3.30.*

```
        MOV AX,offset RETURN    ; get return address' offset
        PUSH AX                 ; push flags and return address
        PUSH CS                 ; onto stack in reverse order
        PUSHF                   ; save flags for IRET
        MOV CL,9                ; display DOS string
        MOV DX,offset MSG       ; this is the message
        PUSH CS
        POP DS                  ; verify that DS = local code
        JMP dword ptr ALT_DOS_PTR; and execute the function

RETURN:
        MOV AH,4Ch              ; terminate a process
        INT 21h                 ; via DOS

ALT_DOS_PTR  DW 00C0h,0000      ; entry point for alternative
                                ; DOS handler (0:00C0h)

MSG       DB  0Dh,0Ah,"Example of backdoor MS-DOS "
          DB "function call.",0Dh,0Ah,7,"$"
```

Example 2: *A far jump executes the call and the dispatcher returns to the stack.*

```
        MOV AX,offset RETURN    ; get return address' offset
        PUSH AX                 ; push flags and return address
        PUSH CS                 ; onto stack in reverse order
        PUSHF                   ;
        MOV CL,13h              ; DELETE FCB function
        MOV DX,offset FCB       ; this is the special FCB
        PUSH CS
        POP DS                  ; verify that DS = local code
        JMP dword ptr ALT_DOS_PTR; and execute the function

RETURN:
        MOV AH,4Ch              ; terminate the process
        INT 21h                 ; via DOS
ALT_DOS_PTR  DW 00C0h,0000      ; entry point for alternative
                                ; DOS handler (0:00C0h)

FCB      DB 0FFh                ; extended FCB
         DB 5 dup(0)            ; reserved bytes
         DB 1Fh                 ; all attribute bits set
         DB 0                   ; default drive ID
         DB "???????????"       ; match all files
         DB 19h dup(0)          ; rest of FCB
```

Example 3: *An FCB function can delete files. Calling this function while at the root directory will obliterate all files on the disk, requiring tedious work with your favorite disk editor to restore them.*

lems, and that the alternative DOS handler is limited in its scope, how serious a danger do these backdoors present? Only the standard input/output and FCB functions are allowed, and although FCBs can delete and rewrite files, they can be used only on files in the current directory. Although a Trojan Horse program could use these backdoor approaches to do some damage, it would not seem to pose a major problem.

This would be true, except that one FCB call, function 13h (DELETE AN FCB), has a special case that could destroy all files on a hard disk. The special case requires that an extended FCB use a filename of "??????????" and an attribute of 1Fh. Seeing this specific combination, function 13h deletes all files in the current directory, including files marked with the read-only, volume, and subdirectory attributes. To make matters worse, this function replaces the first character of the deleted filenames with a 0, not the usual 0E5h. This prevents most "undelete" utilities from being able to undo this call's severe damage.

Consider the potential damage of this call. If executed at the root directory, it effectively deletes all files on the disk. As subdirectories are only special files that contain directory information, these are also deleted. This therefore prevents any access to the files that were in those subdirectories, including any deeper subdirectories. Note that the files in the subdirectories are not deleted, and their space remains allocated; only the directory information about them has been erased. CHKDSK will therefore report these orphaned files as being unallocated clusters. It is possible to recover these files and the original tree structure, but only with painstaking work with a disk editor.

This behavior of MS-DOS is truly bizarre. Normally, only MS-DOS's internal routines can update or delete the files marked with the subdirectory attribute. That an FCB function is allowed to delete these files is an unbelievable quirk of MS-DOS. Example 3 shows the use of this special case, using the first DOS backdoor. Warning! If you experiment with this call, please do so only on a floppy disk and not on a hard disk. Calling this function while at the root directory will obliterate all files on the disk, requiring tedious work with your favorite disk editor to restore them. (This warning, by the way, is from my own personal experience!)

This dangerous call, therefore, provides the answer to the question asked above: MS-DOS's backdoors present a severe threat to an unsecured system. Even though an anti-viral program may filter INT21h calls, if it doesn't change the vector at 0:00C0, it is easy to destroy all files on a hard disk.

Listing One, page 98, shows one approach with a device driver called BACKDOOR.SYS. By installing the device driver in the first line of your CONFIG.SYS file (DEVICE=BACKDOOR.SYS), you ensure that it is installed before any other programs can run. BACKDOOR.SYS simply replaces the vector at 0:00C0h with a pointer to a new handler and then installs itself as a character device. The new handler refuses any requests for DOS services through the alternative DOS function dispatcher. This effectively closes both of MS-DOS's backdoors. It also filters INT21h to specifically look for the special function 13h call and rejects the function request if it occurs.

No IBM PC or compatible running in real mode can be completely safe from destructive programs, whether intentional or not. However, it makes no sense to allow known dangers to continue to exist. Closing DOS's backdoors removes one of the more obscure dangers to your computer and its data.

DDJ

(Listing begins on page 98.)

Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

RAM Disk Driver for Unix

Reduce overhead and improve performance

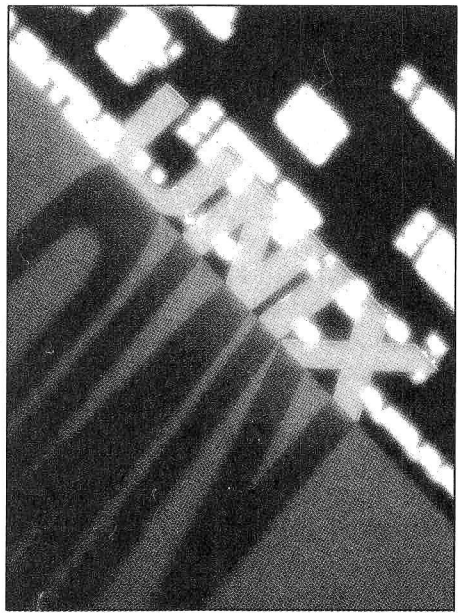
Jeff Reagen

Disk operations are generally slow and expensive compared to other operating system functions. In Unix, performance is boosted by caching the most recently used disk blocks within a pool of RAM buffers.

This pool, called the "buffer cache," utilizes cache hits to complete disk requests without having to access the disk. Unfortunately, the buffer cache is static in size, and eventually some buffers must be written out to disk before they can be reused for another disk block.

In some respects, the Unix buffer cache is a RAM disk. The difference is that the buffer cache manages the reuse of disk blocks, while the RAM disk simply reports the file system as full when all buffers have been allocated.

This article describes the implementation of a RAM disk driver for Unix. A 386 system with four megabytes of RAM running Unix System V/386 Release 3.2 was used throughout to develop the driver.



The Driver Implementation

The RAM disk driver differs from traditional Unix disk drivers in several ways. The first, of course, is the hardware. In reading or writing to a physical disk, the driver must first position the read/write head, then transfer the data blocks. When all is done, typically many milliseconds later, the hard disk presents an interrupt to Unix and the request can be completed. In contrast, RAM disks do not experience the positioning delays associated with mechanical hardware, which eliminates the need for interrupts.

The RAM disk driver developed for this article (see Listing One, page 100) supports both a block-mode and a character-mode interface. Block mode is used by the kernel to read and write mounted Unix file systems. Character mode provides a raw interface to the disk, allowing the application to bypass the Unix buffer cache and manipulate the disk directly. Eight entry points to the driver are provided: *rdinit*, *rdopen*, *rdclose*, *rdstrategy*, *rdread*, *rdwrite*, *rdintr*, *rdioctl*, and *rdprint*.

During the Unix boot, the kernel calls *rdinit*, which determines whether a RAM disk has been defined, and, if it has, allocates the memory required to represent the disk. The *sptalloc* function is used to allocate this memory. Using *sptalloc*, memory is allocated in page-size units (4096 bytes in 386 Unix), and if successful at obtaining the number of pages requested, *sptalloc* returns a kernel virtual address referencing this memory. All page table manipulation is handled by *sptalloc*, including page linkage and the locking down of pages as directed by the PG_P parameter passed to *sptalloc*. Notice the DONT_SLEEP parameter in the *sptalloc* call. Initialization routines in Unix are not permitted to sleep because they can prevent the system from successfully booting.

After space has been carved out for the RAM disk, *rdinit* fills in *rd_cfg*, the configuration structure which represents the state, virtual address, and size of the RAM disk. Of these, only the size field must be defined during driver compile time.

The driver supports three states: RD_UNDEFINED, RD_CONFIGURED, and RD_OPEN. RD_UNDEFINED is the initial state. It indicates the RAM disk has not been set up. RD_CONFIGURED is set by *rdinit* after a successful call to *sptal-*

Jeff is a special projects engineer for Banyan Systems and can be reached at 28 Grant Street, Milford, MA 01757.

(continued from page 50)
loc. RD_OPEN is logically ORed to the state field each time the disk is opened. RD_OPEN could be used at a later time to implement a critical region lock for the RAM disk, allowing only one open at a time.

rdopen is either called when a mount is being performed or when an application attempts an open on the device node associated with the RAM disk.

This RAM disk driver supports both block and character mode interfaces

Application programs use the 8 bits of the minor number to inform the driver which disk should service the request. In this implementation, *rdopen* recognizes only one RAM disk, and rejects the request if it specifies any disk other than zero. Rejected requests are handled by updating the error field in the current *u* area (defined in "sys/user.h") and returning to the caller. An error is returned in the same fashion if the state maintained by the configuration structure *rd_cfg* indicates the disk has not been initialized. With request verification out of the way, *rdopen* turns on the RD_OPEN state bit.

rdclose is simple since there is no mechanical hardware associated with a RAM disk. *rdclose* simply turns off the RD_OPEN state bit. You may think this is an error because we are turning off the open bit even though multiple opens may have been performed on the device. Not to worry; if a file has been opened multiple times in Unix and a close request is then made by one instance, Unix simply decrements in internal file open count. When that count reaches zero, the close function is called.

rdstrategy is the driver function that actually services the read/write requests. A Unix buffer header (described in sys/buf.h) is passed as a parameter containing all necessary information to service the request. Pertinent information such as the requested block, number of bytes to transfer, direction of transfer, and the target device are filled in by the kernel prior to calling *rdstrategy*.

Rather than trusting all information presented by the buffer header, *rdstrategy* performs some simple error checking; without error checking, a simple request exceeding its limits could crash the system. *rdstrategy* begins by converting the requested block number into a byte offset which is used to reference the start of the request. The number of bytes requested is then added to this start location to determine if the request will go off the end of the disk. If the operation is a read request, this behavior is tolerated so end of file (EOF) can be detected by the application. The request is adjusted so all data up to and including the last block is read. However, write requests are rejected immediately.

Now that the error checking is out of the way, *rdstrategy* can transfer the data into or out of the RAM disk. Since this is nothing more than a copy operation, *bcopy*, which is supplied by the kernel, can be used. *bcopy* is used to copy a specified number of bytes from one location of kernel memory to another.

Upon completion of the copy, the residual count and error fields of the buffer header are updated to indicate the request has been serviced successfully. *iodone* is then called to return the buffer to the process responsible for initiating the request.

An application or system administration utility references the character interface through *rdread* and *rdwrite*. These routines are virtually identical except for the direction of the data transfer.

Unix provides two support routines, *physck* and *physio*, which are used to transform the request into a buffer header suitable for *rdstrategy*. In a manner similar to *rdstrategy*, *physck* verifies that the submitted request is within bounds of the RAM disk. If the request is a write and it exceeds the bounds of the disk, *physck* returns an error. However, in order to handle EOF correctly, read operations must be trimmed back and the transfer count adjusted so the request will not exceed the last valid disk block. The *physio* function performs all housekeeping chores such as extracting information from the user area to build a buffer header, preventing user buffers from being paged out, and finally calling *rdstrategy* to get the request serviced.

The *ioctl* Unix interface is a repository for miscellaneous driver functions. The RAM disk driver supports only one *ioctl* function, which returns information about the size of the specified RAM disk. The size information is taken out of the *rd_cfg* structure and returned to the process initiating the request. The buffer location passed into *rdioctl* is not in the same memory map as the *rd_cfg* structure. To get around this

(continued from page 52)
problem, Unix provides the *copyout* function. *copyout* allows a driver to copy data out of kernel space and into data structures residing in user space. Example 1 illustrates how an application program would query the RAM disk about its size.

An interrupt handler is not required for the RAM disk, but one is supplied (*rdintr*) just in case some spurious interrupt is dispatched to the RAM disk. In that case, a warning message reporting the spurious interrupt is sent to the console via *cmn_err*.

Information or messages that refer to the RAM disk are dealt with by *rdprint*. For example, if the RAM disk has no free blocks left to allocate and a request for RAM disk blocks arrives at the file system, it calls *rdprint*, passing along a text string stating that the RAM disk is out of space.

Adding the RAM Disk to Unix

After the driver has been compiled, copy the object file to */etc/conf/pack.d/rd/Driver.o*. The kernel must now be rebuilt in order to pull in support for the RAM disk driver. There are numerous ways to rebuild a kernel and complete necessary installation procedures. The method described here utilizes the Installable Driver Package (IDP), which is used on most Unix System V/386 Release 3.2 systems. Throughout the build process, *rd* is the prefix used to uniquely identify the RAM disk driver.

To begin, a system device file must be created that describes what hardware resources the RAM disk requires. Example 2(a) shows the file used for this article.

Because hardware resources are not required for this driver, the interrupt level, interrupt vector, I/O address, and memory address fields all contain a zero. The first three fields tell the build process the name of the driver, confirm that the driver is to be linked into the kernel, and report the number of devices supported, respectively. Copy this file to */etc/conf/sdevice.d/rd* prior to relinking the kernel.

Next, an entry must be added to the

```
#include "sys/types.h"

main ()
{
   int  fd;
   struct rd_size {
      daddr_t sector_count;
      long    b_count;
   } ram_disk_size;

   if ( (fd = open ("/dev/rdsk/rd0", O_RDONLY)) < 0)
   {
       printf ("Could not open RAM disk to do ioctl.\n");
       exit (1);
   }
   if ( ioctl (fd, RD_GETSIZE, &ram_disk_size) < 0)
   {
       printf ("Could not determine size of RAM disk.\n");
       exit (2);
   }
   printf ("The RAM disk consists of %d sectors occupying %d bytes.\n",
      ram_disk_size.sector_count, ram_disk_size.b_count);
}
```

Example 1: *How an application queries the RAM disk's size*

master device file that describes the driver interface. This is shown in Example 2(b).

The first field says this description applies to the RAM disk driver. Next, "ocrwiI" indicates the RAM disk supports an *open*, *close*, *read*, *write*, *ioctl*, and *init* interface. The third field describes the driver as installable, capable of supporting character and block devices, and containing only one entry in the system device file described previously. The fourth field is the handler prefix used to distinguish the interface entry points. Fields five and six are for block and character major numbers. Setting these fields to zero lets the build process assign these numbers dynamically. Fields seven and eight specify the minimum and maximum number of units that can be declared in the system device file. The last field is for Direct Memory Access (DMA), and, since this device doesn't use any, this field is assigned –1.

I've found the easiest way to add this information to the master device file is to create a file in your local directory called "Master," containing the line in Example 2(b). Then use the *idinstall* command listed in Example 2(c) to append the description to the master device file.

The command line informs *idinstall* that the description in file Master is to be added to the master device file; upon completion, do not delete file Master.

After running *idinstall*, the major numbers assigned by *idinstall* can be extracted from the master device file in */etc/conf/cf.d/mdevice* and used to confirm the major numbers assigned to the block and character device files by *idmknode* during the next system boot.

To have the boot process automatically generate the RAM disk device nodes, a node file is needed. Example 2(d) describes two nodes, one for the block and one for the character device.

Given the configuration in Example 2(d), the block device will be addressed as */dev/dsk/rd0* with minor number 0, while the character device is referenced as */dev/rdsk/rd0* using min[illegible] number

Example 2(a):

```
rd    Y   1   0   0   0   0   0   0   0
```

Example 2(b):

```
rd    ocrwiI  icbo    rd  0   0   1   2   -1
```

Example 2(c):

```
/etc/conf/bin/idinstall -a -m -k rd
```

Example 2(d):

```
rd   rdsk/rd0    c   0
rd   dsk/rd0     b   0
```

Example 2(e):

```
cd /
# Make a filesystem on the RAM disk.
/etc/mkfs /dev/dsk/rd0 2048:150
/etc/mountall /etc/fstab
```

Example 2: *Examples for developing Unix RAM disk*

(continued from page 55)
0. Modifying the driver to support multiple RAM disks would require additional entries in Example 2(d) as well as unique minor numbers. To complete the node setup, copy the file described in Example 2(d) to */etc/conf/node.d/rd*.

Before proceeding, it makes sense to make a backup copy of the kernel just in case the new driver has introduced a bug and prevents the system from booting. Then, if there are problems, the old kernel can be used.

At this point, all support files are in place and the kernel can be rebuilt by issuing */etc/conf/bin/idbuild*. If all goes well (meaning all external references were resolved) the current kernel can be shut down and the new one brought up for driver testing.

Automating the Installation

RAM disks represent a form of volatile storage. Because everything written to the disk is lost during a system shutdown, the disk must be initialized every time the system is restarted. Rather than manually issuing the proper commands each time, a system administrator could automate the procedure by adding special commands to the Unix startup scripts.

To begin with, a mount point must be established so the user community can reference the RAM disk. To create */ramdisk* as the mount point use the following *mkdir* command: *mkdir/ramdisk*.

The mount point needs to be created only once as long as you don't delete it after unmounting the file system. Next, append the following line to */etc/fstab*: */dev/dsk/rd0 /ramdisk*.

The */fstab* file contains all file systems to be mounted during the Unix boot. The last step involves building the file system on the RAM disk so the boot procedure can find something to mount. The file system can be created by adding a *mkfs* command to */etc/rc2.d/S01MOUNTFSYS*. Example 2(e) illustrates a modified S01MOUNTFSYS file.

It's important to note that the *mkfs* command is executed before directing */etc/mountall* to read and mount all file systems listed in */etc/fstab*. In Example 2(e), the *mkfs* command is instructed to build a file system on the RAM disk using 2048 sectors (each sector is 512 bytes in size) and allow a maximum allocation of 150 files. Overhead required by the file system is minimal. However, the actual number of *inodes* and *filesystem* blocks will be reduced to 144 and 2024, respectively.

Although the procedure above will automate the installation of the RAM disk, it should be noted that this is a static configuration. For example, reconfiguring the RAM disk to increase its size from one megabyte to two requires the *mkfs* command buried in */etc/rc2.d/S01MOUNTFSYS* to be modified so the additional space provided can be utilized.

Improving the Performance of Unix

The RAM disk has several practical uses in Unix. For example, many programs rely on temporary files and tend to create them in */tmp*. Instead of using part of the root file system for */tmp*, try mounting the RAM disk on */tmp*. This can be done by simply editing the */etc/fstab* and replacing the */ramdisk* mount point with */tmp*.

Another possibility is to reduce overhead associated with loading files. This is a matter of identifying popular files and copying them to the RAM disk. Make sure the PATH environment variables are updated so the RAM disk is searched for the file first.

If your system allows for more than one swap device, a significant performance gain can be had by making the RAM disk a primary swap device. The swap device is the area of disk used by the buffer cache when it becomes necessary to page out buffers. In a heavily loaded system, the RAM disk could keep the system from thrashing itself to death because the swap area no longer has to wait for a slow disk.

Conclusion

Writing a RAM disk driver is a great way to get your feet wet in Unix device drivers. The driver developed here supports only one RAM disk, one megabyte in size. It's a trivial exercise left to the reader to change the size of the RAM disk or extend the driver so it may support additional disks. This driver should port to other flavors of Unix with minimal effort.

Source Code Availability

As a service to our readers, all source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Drive, Redwood City, CA 94063, or call 800-356-2002 (inside Calif.) or 800-533-4372 (outside Calif.). Specify issue number and disk format. Code is also available through M&T's Telepath online service (via TYMNET) and through the DDJ Forum on CompuServe (type GO DDJ).

DDJ

(Listing begins on page 100.)

Vote for your favorite feature/article.
Circle Reader Service **No. 4.**

Optimal Determination of Object Extents

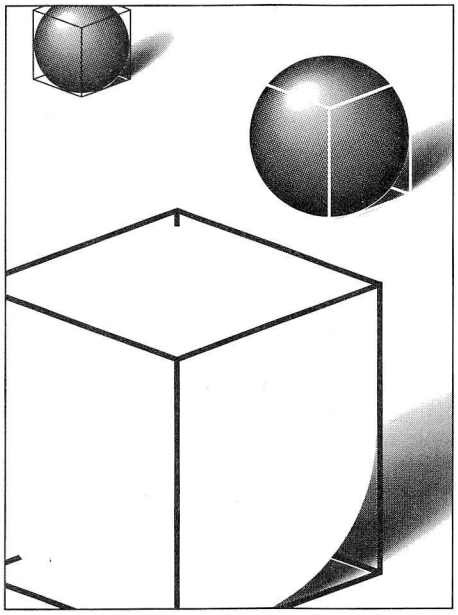
Find min *and* max *the best possible way*

Victor J. Duvanenko,
Ronald S. Gyurcsik, and W.E. Robbins

Extents are often used in computer graphics and constructive solid geometry where the edges of the smallest, axes-aligned rectangle enclosing an object are the extents of that object. In two-dimensional space, this is also called "boxing" and in three-dimensional it is a "three-dimensional (3-D) rectangle" — a parallelpiped.

Graphical objects are often defined by a set of vertices (points) in three dimensions. Finding extents of an object defined by a set of points is a simple process once you realize that the extents are the minimum and the maximum of the object in every dimension. To find the minimum (or maximum) X coordinate, you must search the vertex list that defines the object. The well-known algorithms *find_min* and *find_max* are used to accomplish this (see Listing One, page 102). The procedure is then repeated for Y and Z dimensions.

If an object is defined by N vertices, this approach takes (N −1) comparisons to find the X_{min}, (N −1) comparisons to find the X_{max}, and so on, for a total of 6 (N −1) comparisons for a 3-D object. However, is this the minimal number of comparisons? It's been shown that (N −1) comparisons are necessary to determine the minimum (or the maximum) of N items.[1] In other words, the algorithms *find_min* and *find_max* in Listing One are optimal and it is not possible to do any better by comparison of items! If you were to perform fewer than (N −1) comparisons, an incorrect answer would result for the same input. Yet, it is possible to perform fewer comparisons if both the minimum and maximum are needed, as is the case with object extents.



When minimizing the amount of work done by two separate operations — searching for minimum and maximum in our case — the usual technique is to look for any work or intermediate results that can be shared. In this case there does not seem to be any shared work; the *find_max* procedure compares all of the points to *max* and the *find_min* procedure compares all of the points to *min*. In many cases it may be beneficial to sort the list of items, or a sublist of items. Sorting the whole list makes things worse, because sorting is a slower order operation, $O(N\log N)$, than finding a minimum or a maximum, $O(N)$.[1] However, sorting two items helps minimize the work since this takes only a single comparison.

The strategy used to find the extents is as follows (see the *find_min_max* procedure in Listing Two, page 102):

1. Compare the first two items to each other. Place the larger in *max* and the smaller in *min*.
2. Take the next two items and compare them to each other. Then compare the larger item to *max* and the smaller item to *min*. If the larger item is larger than *max* place it in *max*. If the smaller item is smaller than *min* place it in *min*.
3. Take care of the last item if the number of items is odd. If this item is greater than *max* it can not be smaller than *min*, so there is no need to compare to *min*.

This procedure is slightly more complex, but the gains are dramatic. In fact, to find the minimum vertex and the maximum vertex in the X dimension for an object with N vertices (3N/2 −2) comparisons will be performed by

Victor is a graduate student at North Carolina State University majoring in electrical and computer engineering with a minor in computer graphics. He can be reached at 1001 Japonica Court, Knightdale, NC 27545; or via e-mail victor@ecesting.ncsu.edu.

Machine	MIN/MAX	MIN & MAX	Speed up
PC/AT 8 MHz 600,000 items	34.82 sec.	48.20 sec.	27.8%
VaxStation2000 1,200,000 items	13.70 sec.	18.70 sec	26.7%
DEC 3100 24,000,000 items	17.30 sec.	32.70 sec.	47.1%
Sun SparcStation 24,000,000 items	50.70 sec.	66.60 sec.	23.9%
Cray Y-MP 60,000,000 items	30.21 sec.	43.97 sec.	31.3%

Table 1: *Results of MIN/MAX algorithm*

(continued from page 58)

find_min_max if N is even, and (3N/2 −3/2) comparisons if N is odd. In contrast, 2(N −1) comparisons will be done by *find_min* and *find_max*. This amounts to a savings of approximately 25 percent. What is most important about this result is that it has been shown to be optimal![1] In other words, it is not possible to make any fewer comparisons of items! What's more, the *find_min_max* procedure is general purpose and can be used in many other applications.

Application

As one example of computational savings, let's apply the *find_min_max* procedure to a polygon clipping test. Assume that 100,000, 4-vertex polygons need to be displayed per second.[2] Each of the polygons must be tested to see if it falls completely inside of the display window's boundaries which are 3-D boundaries. Each polygon must be handled separately since no relationship can be assumed between polygons in general. There are three possible ways to handle this:

1. The vertices of each polygon could be tested to see if all of them fall within the 3-D display window. This implies that 400,000 vertices must be checked against two boundaries in each dimension (for example, *window_x_min* and *window_x_max*), or six comparisons per vertex. In other words, this requires 2,400,000 comparisons per second, or 2.4 Mflops.[2]
2. Extents (*min* and *max*) of each polygon could be found and compared to the window boundaries. This implies 2(4 −1) comparisons per dimension, plus a comparison of *min* to the window minimum and of *max* to the window maximum — a total of eight comparisons per dimension. Therefore, 24 comparisons per polygon would have to be made, or 2.4 Mflops (as before).
3. Perform (3*4/2−2) or four comparisons per dimension plus the two comparisons to the window boundaries, for a total of six comparisons per dimension. Therefore, 18 comparisons would be made per polygon, or 1.8 Mflops. This is a savings of 0.6 Mflops, or 25 percent of 2.4 Mflops.

This example demonstrates that valuable computing resources are being wasted by performing redundant work using the first two methods.

Benchmarks

The MIN/MAX algorithm was benchmarked on several computer systems using lists of floating-point numbers of various lengths. The results are summarized in Table 1. Note that the MIN/MAX algorithm delivers more than the promised 25 percent speedup, on the majority of the systems tested. Since the algorithm performs fewer comparisons, it performs fewer stores (writes) of items to *min* and *max*. In fact, the number of writes is potentially reduced in half, as each item is compared to either *min* or *max*, but never both, in the MIN/MAX algorithm. This may reduce bus traffic in some systems.

The method described is applicable to computational environments where a comparison is a binary operation. Most of today's sequential software environments satisfy this condition. However, it is possible to build hardware to find the *min* and the *max* of a fixed number of items in a single step.

References

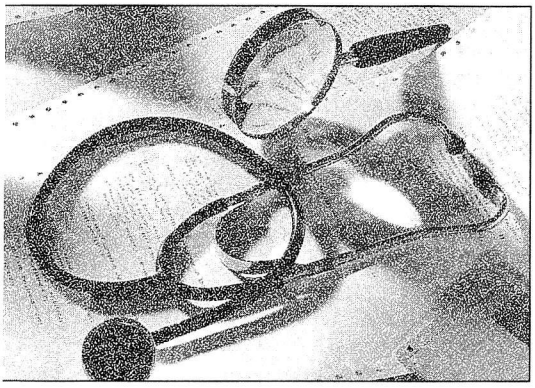
1. S. Baase. *Computer Algorithms: Introduction to Design and Analysis*, second edition, Reading Mass:, Addison-Wesley, Nov. 1988, pp. 24–26, pp. 125–128.

2. K. Akeley and T. Jermoluk. "High-Performance Polygon Rendering," *SIGGRAPH, Computer Graphics*, vol. 22, no. 4, August 1988.

DDJ

(Listings begin on page 102.)

Vote for your favorite feature/article.
Circle Reader Service **No. 5.**



Unraveling Optimization in Microsoft C 6.0

Selecting the right switch for the right job

Bruce D. Schatzman

Optimizing compilers can be both a blessing and a curse. Certainly, we are all happy to have our code automatically transformed into smaller, faster packages: We spend less time sifting through long source listings to uncover inefficiencies, we can write code in more readable form, and we don't need to resort to assembly language for optimum speed as often.

However, automated code transformation has its drawbacks, including the tendency to decrease diligence on the part of some programmers. There is a temptation to get sloppy because "the compiler will take care of it." The compiler does, indeed, handle many inefficiencies, but not all of them. Understanding exactly what a compiler will do to your code (and what it will not do) is an important step toward producing software that displays both speed and readability. In short, optimizing compilers are a supplement to hand optimization, not a substitute for it.

This article focuses on both the practical and theoretical aspects of code optimization, using Microsoft's recently introduced C Version 6.0 compiler (C6) as the tool.

You usually have several goals in mind when adopting an optimizing compiler. Among them may be to:

- Decrease the number of processor cycles required by the program.
- Decrease the number of address calculations performed.
- Minimize the number of memory fetches.
- Minimize the size of the object code.

In all optimizing compilers, most of these goals are achieved primarily through "local" optimizations. Local optimizations are improvements to small, mostly self-contained blocks of code such as loops, switches, and if-else constructs. Despite all the hype from compiler vendors on "global optimization," most code transformations are still performed at the local level. Typical local optimizations include elimination of common subexpressions and dead variables, movement of loop-invariant computations, and folding of constants into a single value.

In general, local optimizations are fairly easy to perform, mostly because they are easy to detect and implement. Compilers have little trouble fixing such local inefficiencies because they are reasonably independent of the program's flow of control on a macroscopic level. The behavior of variables and expressions within small blocks is, for the most part, predictable; the number of possible values taken by these items, as well as how they fit within the flow of control, can be understood.

However, programs contain more than just a series of isolated blocks, and until recently, commercial compilers made almost no effort to optimize across entire routines. In an attempt to produce still more efficient code, compilers are now appearing on the mar-

(continued on page 66)

Bruce has worked in the computer industry for more than ten years, holding a variety of technical and marketing positions at corporations including General Dynamics, Tektronix, and Xerox. He is currently an independent consultant and can be contacted at P.O. Box 5703, Bellevue, WA 98006.

Switch	d	l	c	g	e	eg
Listing						
One	385.3	234.2	279.3	215.7	239.9	189.3
Two	312.7	228.0	241.1	208.0	207.4	179.5
Three	231.2	167.8	172.6	164.9	149.0	145.4
Switch	**lg**	**le**	**leg**	**x**	**a**	**al**
Listing						
One	258.3	196.2	177.0	180.2	279.3	234.2
Two	259.9	185.6	175.3	181.9	241.1	227.9
Three	203.5	139.7	138.0	140.4	172.5	167.8
Switch	**ag**	**age**	**aleg**	**ax**	**axz**	**zl**
Listing						
One	215.8	189.3	177.0	180.1	180.1	234.2
Two	208.1	179.5	175.3	181.9	181.9	228.1
Three	164.8	145.4	138.0	140.4	140.4	167.9
Switch	**zlg**	**axz G2**				
Listing						
One	258.3	177.2				
Two	259.9	180.8				
Three	203.6	144.0				

Table 1: *The "switch" table*

(continued from page 62)
ket with the ability to perform "global" optimizations. This type of optimization looks at code on a broader scale, seeking inefficiencies that affect multiple blocks. This task is much more difficult to achieve because of the more complex data flow analysis that must be performed by the compiler.

Strategies for Using C6 Optimizations

Although somewhat of a simplification, the "90/10" program execution rule is generally true. That is, 90 percent of a program's execution time is spent in about 10 percent of its code. Optimizing a routine to achieve a 50 percent speed increase sounds impressive, but if your program spends an average of 110 microseconds in that routine for every minute of execution time, the results are hardly noticeable or worth the time to achieve. Simply turning on a variety of optimizing switches (such as with */Ox*) at compile time is likely to yield only modest improvements in speed. A small amount of extra work can give you much better results. These results can be achieved, following the 90/10 rule, by implementing an optimization strategy that focuses only on critical sections of code that represent the greatest amount of "execution drag." Uncovering these critical portions of code can be done with a good profiling tool. Unfortunately, C6 does not provide such a tool. For this article, I used Inside! profiler (Paradigm Systems, Endwell, N.Y.) to produce a number of useful execution statistics.

Paradigm's profiler is available in a number of versions that are compatible with debug information and map files generated by C6/QuickC, Turbo C, Zortech C++, and other non-C compilers. This tool has several strengths, including the ability to provide timing statistics at the function, block, and source-line levels. Some profilers can identify which functions are consuming the most cycles, but will provide no insight into which elements within a function represent candidates for optimization. Block- and line-level analysis enables the programmer to identify exactly where cycles are being consumed within a function, eliminating guesswork and saving time.

Inside! also has the ability to isolate time spent during DOS calls, enabling you to measure the impact of servicing various DOS interrupts. It is often the case that DOS I/O represents the largest percentage of execution time within a function. For example, if a loop processes 100 strings and writes each to a file using *fputs( )* on every loop iteration, there is no question that this con-

sumes an enormous amount of cycles. A much better strategy would be to buffer all 100 strings in memory during the loop and write them as a block using a single *fputs( )* function after the loop completes. Optimizations such as this can yield execution times that are orders of magnitude faster. Profilers make optimizations such as this more obvious.

For maximum performance, the following strategy is recommended, assuming your code is already in a reasonably stable state:

1. Disable all optimizations on your compiler (*/Od*) and produce an executable.
2. Use a profiler to determine which functions consume the majority of cycles. Most profilers provide a "percentage of total runtime" value for each function. Record all statistics in a file.
3. Recode the major algorithms in these functions (if possible) for more efficiency, and make local and global optimizations (such as elimination of redundant subexpressions and unfolded constants) by hand.
4. Declare all major functions with the *_fastcall* keyword, and use based pointers for access to far code and data. Elimination of aliasing, if possible, will enable more aggressive compiler optimization when you get to that stage.
5. Produce another executable, again with compiler optimization disabled, and run your profiler. Compare the new statistics to your previous values to measure progress.
6. Compiler optimizations can now be used. Place pragma-level switches in those files containing the cycle-hungry functions, and/or use option file-level switches where appropriate.
7. Run the program several more times under control of the profiler using different combinations of pragmas and/or switches to determine which yield the best results. Save all profiler statistics, and compare these with the original statistics to ensure that the compiler is helping, not hurting. A "switch table" (see Table 1) will help you record your times.
8. Optimize all of the minor routines of the program using only file-level switches or pragmas. Hand-optimization of these minor routines will probably not be worth the effort.

Although this approach may seem overly thorough, it usually yields the best results. Step 3 might seem to defeat the purpose of an optimizing compiler, but there is no amount of compiler optimization that can compensate for a poorly designed algorithm. Typi-

cally, the best that can be expected from an optimizing compiler is a 10 – 50 percent improvement in speed for well-written code. Recoding an algorithm may boost performance many times this amount.

The Role of Aliasing in Optimization

Most compilers have trouble dealing with aliased variables. Using aliases will inhibit most of the optimizations that could potentially be performed — and there are many. The */Oa* switch is one of the most effective optimizations because it enables many local code improvements that otherwise could not be done.

Consider the code in Example 1. In this example, **p* is an alias of the variable *x*. In MS C5.1, if the */Oa* option is used during compilation, the compiler will assume that no variables have been aliased (which is not true). The expression *x+y+z* will be seen by the compiler as being constant (or "loop-invariant") for the duration of the loop. There is thus (apparently) no need to compute *x+y+z* 100 times, and it will be hoisted out of the loop and replaced with its assumed constant value. However, the assignment of *k* to **p* (which is really *x*), means that *x+y+z* is not constant, and the program's results will be incorrect when executed. This problem would not have happened if aliasing had been declared (*/Oa* is not specified), because subexpressions are not factored (hoisted) from loops in this case.

```
void main(void)
{
        int  x, y, *p;
        p = &x;
        x = 1; y = 2; z = 6
        for(k = 1; k <= 100; k++) {
                                        k += x + y + z;
                                        *p = k;

        }
}
```

Example 1: *p *is an alias of the variable* x.

Types of Optimizations Provided by C6

Among the more important optimization features in Microsoft C 6.0 are pragma-level optimization, global optimization, new peep-hole optimizations, based pointers, and in-register parameter passing (*_fastcall*). With the exception of based pointers (which are described in *DDJ*, August 1990, page 85), I'll briefly discuss each of them.

Pragma-Level Optimization

In C 5.1, some optimizations had to be disabled because the behavior of the optimized code could be unpredictable. For example, imagine a program file that consists of three functions, the first of which uses aliased variables. In this case, you must disable alias-sensitive optimizations across all three functions, and the ability to optimize the two functions which do not use aliasing is lost.

In C6, you can fully optimize the two non-aliased functions while still preventing alias-sensitive optimizations within the first function. This is accomplished by placing new "pragma" statements within the program module.

Pragmas are not optimizations per se, but are control directives parsed by the compiler. With pragmas, virtually all optimizing switches can be toggled on or off at the function level. You can also use these constructs to toggle combinations of switches such as *#pragma optimize("lec", off)*, which will turn off loop optimization, global register allocation, and reduction of common subexpressions.

Of the many new optimization-related enhancements offered by C6, pragmas could have the greatest overall impact on program performance. They allow a finer level of control over the compiler than was possible in C5.1, and take greater advantage of the programmer's own knowledge of the code.

Global Optimization

With C6, programmers can use the */Oe* switch, which tells the compiler to manage register variables on a function-wide basis. When this switch is specified on the command line (or in a pragma), the frequently used *register* keyword is ignored, and the compiler takes over. Although it is a good bet that this switch will result in speed improvements, it is a mistake to assume that the compiler can always do a better job of register-variable allocation than you can. For functions that already declare register variables, use a good profiling tool to measure the program's performance both with and without the */Oe* switch in order to verify that performance really does improve. The same advice applies to any optimizing operations — verify to be sure. Another C6 global optimization factors common subexpressions across entire functions. The corresponding switch is */Og*, which differs from its predecessor (*/Oc*) by searching for such subexpressions across entire functions. Consider this line of code: *x = y[k] + (c * d + 100 * e);* If the subexpression (*c * d + 100 *e*) is found to exist within two loops, three if-else constructs, and two switch-case blocks, it will not be factored 2 + 3 + 2 = 7 times, but just once. This, of course, is if the subexpression is completely invariant. If it is variant, it will be factored by the number of variances encountered.

Peephole Optimizations

In the world of compilers, a peephole optimization is simply a specialized improvement on a small (but potentially important) piece of code, typically just a few instructions. This is where assembly language expertise pays large dividends. Peephole optimization is performed by looking for special code patterns that may be replaced with more efficient equivalents. A simple example is "constant folding" which can be illustrated with this line of code: *x = 12*(y + 25) – z*(y + r + 9 + 20) + 10;*

The three computations which can be performed by the compiler before the program is actually run are: *12 * 25 = 300*, *9 + 20 = 29*, and *300 + 10 = 310*. A good compiler will replace this code with its more efficient equivalent: *x = 12*y + 310 – z*(y +r + 29);* This optimization saves one multiply and two adds. In a loop of 100 iterations, this amounts to 100 multiplies and 200 adds, not to mention the associated register operations. C6 in-

C6 does a much better job of handling aliased variables. In the code sample, even if the programmer specifies */Oa* at compile time, the C6 preprocessor will notice that the address of *x* has been taken and will assume that its value might not be constant within the loop. Because *y* and *z* have not had their address taken, the compiler assumes that these variables are constant, and will remove them while leaving *x* correctly within the loop. C6 therefore makes the */Oa* option safer than its former implementation in C5.1.

In some cases, aliasing can occur without the explicit declaration of the address operator (such as when two pointers are equated), and the preprocessor will not detect it. However, if an option is selected that invokes the global optimizer (such as */Oe*, */Og*, or

cludes bit-shifting and switch operations. Code blocks having these two types of statements should run faster in C6 than in C5.1.

In-Register Parameter Passing

To minimize the large amount of memory I/O incurred by a function call, C6 includes the *_fastcall* convention which generates faster code by passing selected arguments through registers, rather than pushing and popping from the stack. *_fastcall* is a "strongly typed" calling convention, meaning that the register selected for a particular variable is related to the variable's data type (which defines its size).

_fastcall produces different results depending on the function. Functions passing only a few characters and/or integers will probably see the greatest speed increases because all parameters can be passed via registers. You control which functions should use the *_fastcall* convention by placing the keyword in the function declaration. For example: *int_fastcall func(int, long, long, int, long);* This feature can also be implemented on a file-wide basis with the */Gr* option, which converts each function in your file to the *_fastcall* convention. It should be noted that *_fastcall* cannot be used with functions having variable-length argument lists, nor can it be used with functions containing *_asm* blocks. —B.D.S.

/Ol), the optimizer itself will perform a data flow analysis that can detect aliases which occur without the address operator. Thus, switch combinations such as */Oae* and */Oal* represent an additional layer of safety above */Oa*.

It should be mentioned that the "a" and "z" options have been eliminated from */Ox* in C6. Microsoft explains that this step was taken to increase the safety of */Ox*. For some code, */Ox* might actually produce slower execution times in C6 than in C5.1. However, */Ox* will also invoke the new global optimizer, meaning that execution times may still be faster in C6, even with the elimination of "a" and "z." Simply specifying */Oxaz* will include these two switches during compilation.

Another step Microsoft has taken to deal with the aliasing problem is the new */Ow* switch, which tells the compiler to assume no aliasing except when calling functions (which typically pass pointers). This switch flushes all variables held in temporary locations (such as registers) back to memory before each function call, producing only one instance of a variable that can be modified by a function. This prevents the assignment of two different values to the same variable — one in memory and one in a register.

The Effectiveness of Compiler Switch Combinations

Considering the wide variety of switches and options in today's compilers, how do you know which switches are appropriate for a particular program? Some may yield good results, while others are disappointing. Because of the infinite variety of programs, there is no chart that indicates which combination of switches should be optimal for your software. However, there are a number of rough guidelines that can be followed. Some sample code will help illustrate the process.

Consider Listings One through Three (page 104). These are all different forms of a program that approximates the definite integral of three functions and prints the results. The three functions are:

$$y(x) = 2x^2 + 46x + 10$$
$$y(x) = 3x + 54$$
$$y(x) = x^3 - 72x^2 + 14$$

The definite integral may be thought of as the area between the curve of the function (as plotted in the *xy* plane) and the *x* axis. This particular program measures the area for the interval between $x = 0$ and $x = 100$. The area is approximately equal to the sum of 100 rectangles, each of length 1 and height $y(x)$, where x is the midpoint of the base of each rectangle.

Listing One was designed to test several optimization features. The bulk of the test is provided by the *compute_areas* function, which contains the following challenges:

- There are more register variable candidates than there are registers.
- There are several subexpressions (such as *a*b*) that are both locally and globally redundant.
- Several constant expressions (such as *a*b+c+d*) are not folded into a single value.
- The program is loop oriented, and some of the inefficiencies center around these loops.

Notice that none of the variables in this listing have been explicitly declared as "register." The algorithm itself is also inefficient. Three of the loops can really be combined into one, and there is no need to sort the array elements in order to compute an integral approximation. In addition, two of the loops are repeated 100 times in order to provide more significant digits in the output from the profiler. They are not meant to test the compiler.

Listing Two is an attempt to anticipate compiler optimizations and perform them by hand. None of the algorithmic inefficiencies are fixed. Listing Three combines the improvements made in Listing Two, and makes one

algorithmic enhancement —two of the loops are combined into one.

Table 1 lists the execution times for different combinations of switches on each code listing. The test was performed on a Compaq 386/20 with no floating-point hardware (which would not have been used anyway on this integer-based code). Test results are in milliseconds, and the accuracy of the profiler is within 10 microseconds (+/– .01 millisecond). The last digit was rounded to the nearest hundredth of a millisecond.

In most cases, the test results were predictable, but there are some surprises. The predictable results are as follows:

- On the average, the optimizations performed by hand in Listing Two did not help much when it came to execution time. At least this proved that the compiler did its job well in Listing One, and that certain hand optimizations are a waste of time.
- Loop optimization (*/Ol*) almost never improved performance when used in tandem with *g* or *c*. Because the major problem with this program's loops is redundant subexpressions, loop optimizations are effectively equivalent to reduction of common subexpressions.
- Specifying */G2* (use 80286 instruction set) did not improve performance. I suspect that this was because the generated code was fairly simple and handled equally well by the 8086 instruction set.
- The */Og* switch (global subexpression reduction) was much more effective than */Oc* (local subexpression reduction). For this code, *c* optimizations were a subset of *g* optimizations.

The surprises were as follows:

- It is not surprising that the so-called "maximum optimizations" such as */Ox* and */Oaxz /G2* did not produce the fastest times. The surprise is that they were beaten slightly by */Oleg* and */Oaleg*. (I am unable to explain this because */Ox* supposedly includes the */Oleg* optimizations. It may be useful to try */Oleg* on loop-intensive routines to see if it outperforms other optimizations.) Although only two to four percent better than the maximum, this extra performance may be critical in some applications such as animation, where every millisecond is important.
- Although not shown in the matrix, I declared all integer variables as "register" in all three listings and ran the tests. Performance was equal to or worse than the times shown in the matrix for all switches. I am unable to explain this.
- The */Oe* option (global register allocation) was extremely effective. Specifying only */Oe* produced very fast times. Generally, any combination of switches that included option *e* outperformed any other combination of switches without option *e*.

These code listings are isolated examples. Although this was not a benchmark, C6 performed admirably, providing up to a 54 percent increase in performance over unoptimized code. Different code will undoubtedly give different results, and experimentation is important.

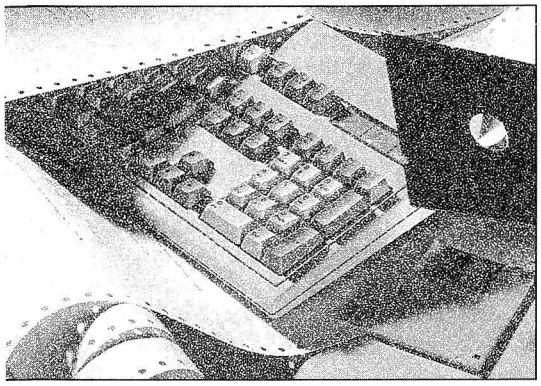
Conclusion

C6 is much more than just a code generator. It is a powerful tool that requires a certain amount of skill to use effectively. The popular technique among C5.1 programmers of consistently using the */Ox* option (which enables many optimizations at once) has less merit in C6. The point is that optimization is the responsibility of both the compiler and the programmer. Programmers who understand that compilers are really a supplement to hand optimization, rather than a substitute for it, will benefit the most. Experimentation with these new optimizing features may be time consuming, but in the end, it is very much worth the effort.

DDJ

(Listings begin on page 104.)

Vote for your favorite feature/article.
Circle Reader Service **No. 6.**



Kermit for OS/2 Part II

Wrapping up the port

Brian R. Anderson

Last month, I began a trek that would lead us from the comfortable world of DOS to the magical land of OS/2. At our side was a familiar friend, Kermit, whose job it is to communicate between different machines whether they be mainframe or PC. In Part I, I presented the PCKermit and Shell modules (Listings One through Nine) and the basics of programming under Presentation Manager. Part II provides Listings Ten through Nineteen, covers some of the problems encountered when porting PCKermit from DOS to OS/2, then examines the communications capabilities of OS/2 as well as the low-level screen and video I/O. Finally, I discuss the implement of TVI950 terminal emulation.

I would like to mention that, because of space considerations, not all listings to PCKermit are provided in the magazine. The complete set of listings is available directly from *DDJ* (see "Availability" on page 3 for details).

Also note that Stony Brook has released Version 2.1 of their Modula-2 since this article was written. This new version is the first to "officially" support Presentation Manager. While functionally equivalent to the listings presented here, the source code mentioned above reflects changes introduced by the new Modula-2 version.

Brian is an instructor of computer systems technology at the British Columbia Institute of Technology. He can be reached at 3700 Willingdon Avenue, Burnaby, B.C., Canada V5G 3H2.

Unexpected Problems

It is not possible to write to the *AVio* presentation space while you are executing within a thread unless you create a separate message queue for the thread — accessing *AVio* (or *Gpi*) without a separate message queue is one of the few activities that will bring OS/2 to its knees.

I found that I could do anything else that I wanted to within a thread (such as open a file, read from the file, send to the COM port, receive from the COM port, or write to a file) without trouble. As soon as I tried to output to the *AVio* presentation space, the system (except for the mouse pointer) froze. When the mouse pointer was moved to the area on the screen where the writing should have occurred, the pointer disappeared; move the mouse away, and the pointer reappeared. I found this particularly odd, as writing to *AVio* actually consists of sending the data to a buffer — the contents of the buffer are not actually displayed until a *WM_PAINT* message is detected within the main thread back in the window procedure.

My solution (rather than adding a second message queue) was to write to *AVio* only while executing in the main thread. For other threads to display information, they had to post messages to the main thread. The main thread then output to the *AVio* presentation space for them.

One severe limitation of OS/2 is that the buffers for the serial ports are fixed size. In other words, there is no way that an applications programmer can specify a larger buffer. I found that while in terminal emulation mode (on an IBM PS/2, Model 70), PM is unable to keep the screen updated when characters from the serial port are arriving at 960 per second (9600 bps). I assume PM message-passing overhead causes this, as no such problem exists while running a similar, but character-based, OS/2 program.

At 9600 bps, the host sends a complete screenful (up to 2000 characters) in about two seconds. However, because the window procedure is advised of each character via a message (similar to a *WM_CHAR* message), it takes PM about six seconds to update the screen (again, this is for an IBM PS/2, Model 70). With the fixed serial buffer (currently 1K, but Microsoft doesn't promise *not* to change that later), a buffer overflow can easily result. And that means lost data and a scrambled screen. As I mentioned in the beginning under the module descriptions, I added one layer of variable size buffering to the Stony Brook serial port module. When I specified a buffer of 4K, the overflow problem was eliminated.

The standard OS/2 serial port isn't the only buffer that can overflow in the situation just described — the default message queue size is ten (fine for

(continued from page 72)
messages generated by user interaction, but inadequate when the serial port is causing the generation of 960 messages per second). Even worse than the low default, there "seems" to be an upper limit on message queue size of 1024 — above this, much of the keyboard interface (including keyboard accelerators) simply quits working. Nevertheless, by maintaining a very large buffer (4K) for incoming characters, I found that there were never problems like those just described with the file transfer threads; the PAD and DataLink modules post only one message for each packet received or sent. Even at the highest transmission speed, this results in only about 20 messages every second — well within the ability of the system to handle.

PCKermit can transfer files between my 386-AT clone and my Atari ST at up to 19200 bps. Although OS/2 supports 19200 bps, IBM PS/2 hardware does not — the Model 70 will not work at 19200 bps (with this or any other communications program that I have tried). PCKermit is able to transfer files between the IBM PS/2, Model 70 and the IBM 3083 mainframe at up to 9600 bps.

Terminal Emulation

TERM.MOD (Listing Ten, page 108) provides TVI950 terminal emulation. TERM-.DEF is included in the listings in Part I of this article, as are all the definition files for the modules discussed later. Terminal emulation consists of intercepting keyboard entries and converting them to match what the host expects (for example, the *Insert* code from the IBM PC keyboard is not the same as the *Insert* code for the TVI950 —the software must make the appropriate translation). Control codes coming from the host must be recognized and the appropriate action taken (for example, for TVI950 emulation, receipt of an *ESC** must result in the video screen being cleared). Term makes calls to Screen, the module which contains the actual functions for doing such things as clearing the screen, positioning the cursor, and so on. Also in the Term module is *Dir*, the function that accepts a path from the user, changes to the specified drive and directory, and then displays the requested directory.

PCKermit can transfer files between my 386-AT clone and my Atari ST at up to 19200 bps

The thread used for terminal emulation (that is, Connect mode) is *TermThrProc* (line 1545). As long as the thread is active, this procedure sits in a loop, getting characters and posting *WM_TERM* messages (line 1554). If no character has been received, a call to *DosSleep* returns the balance of this thread's time slice to the system (no busy waiting loops in OS/2!).

When the window procedure detects the *WM_TERM* message, it calls *PutPortChar* (line 1566) with the character obtained from the *LOWORD* of the *mp1*, the message's first parameter. The *PutPortChar* procedure performs half of TVI950 terminal emulation: It recognizes the TVI950 codes and takes appropriate action (clears screen, positions the cursor, or displays a character). Several other procedures help with this task: *Escape*, line 1604, handles the escape sequences; *Cursor*, line 1620, positions the cursor; *InsertMsg* and *MsgOn*, lines 1636 and 1654, respectively, place status messages from the

host in reverse video at the bottom of the screen.

When the window procedure detects a *WM_CHAR* message, it calls *PutKbdChar* (line 1457). This and several other functions perform the balance of TVI950 terminal emulation: They convert various keystrokes (for example, function keys and cursor keys) into the correct TVI950 codes. All of these special codes are actually handled by *Function* (line 1484).

Also contained in the Term module, but not related to terminal emulation, is the *Dir* procedure (line 1375). Besides displaying a directory, this function allows the user to log on to a different drive, or to change to a different directory (both by simply specifying a complete path while requesting a directory). This feature is necessary in a Kermit program, as only a file name must be specified when sending (for example, you cannot send a file based upon a complete path). This is done because not all computers that use PCKermit recognize Unix-style path specifications. The *Dir* procedure consists mostly of code for parsing the path to separate out the drive, the subdirectory, and the file specification.

The Screen Module

SCREEN.MOD (Listing Eleven, page 110) uses the OS/2 *Vio* functions to control the display. *Vio* allows for various attributes (in either monochrome or color). Constants defined starting on line 1694 and variables defined starting on line 1712 control the attributes. The only video attributes that PCKermit emulates are *NORMAL*, *HIGHLIGHT*, and *REVERSE* (that is all that is needed for interaction with the mainframe host). These three attributes can be assigned various colors: PCKermit provides five color combinations. On line 1837, the *ClrEol* procedure uses *VioScrollUp* to clear to end-of-line by scrolling up a portion of screen (filling it with a blank, *NORMAL* attribute character cell). On line 1961, *WriteAtt* uses *VioWrtNCell* to write a character cell with the current attribute at the present cursor location.

SCREEN.MOD uses *AVio* to provide several video functions including: *ClrScr*, *ClrEol*, *GetXY*, *GotoXY*, and *WriteAtt* (which writes a character plus attribute at the current cursor position). Charles Petzold refers to *AVio* as "the easy way out," but says these functions are "too good to give up entirely." I agree on both counts: *AVio* is certainly easier to use than *Gpi*, but it has everything that is needed to provide the screen output for emulating a standard VDT. I make no apology for taking the easy way out!

Working with CommPort

COMMPORT.MOD (Listing Twelve, page 114) is a module that was supplied by Stony Brook. Its purpose is to transmit a byte at a time to the communications ports (COM1 or COM2). The DataLink module (see Listing Seventeen, page 119) calls this module repeatedly. The original module had a couple of problems, the most serious of which was that for receive, it would wait until a character was available rather than returning immediately with status set to "no character" (this causes the system to hang in many situations). I have modified the module extensively to correct this and a few other problems, and to add a more flexible buffering scheme. (See earlier under "Unexpected Problems" for further details.) The fix involved using *DosDevIOCtl* to check the status of the serial port in several cases, and return proper status information to the caller. In addition, there was a small problem with setting up stop bits.

I further enhanced CommPort by adding dynamically allocated buffers. In the original version, calling *GetChar* resulted in just a single character being fetched from the serial port. If *GetChar* is not called often enough, the OS/2 buffer will eventually overrun. In my enhanced version, each call to *GetChar* transfers all of the characters from

Original Stony Brook Modula-2 (very similar to Microsoft C)

```
WinPostMsg (ChildFrameWindow, WM_PAD,
     MPFROM2SHORT (PAD_ReceivedPacket, 0
     MPFROM2SHORT (PktNbr, rSeq));
```

Stony Brook Modula-2 v2.1

```
VAR
   MP1, MP2 : MPARAM;      (* New variables required *)

MP1.W1 := PAD_ReceivedPacket;    MP1.W2 := 0;
MP2.W1 := PktNbr;    MP2.W2 := rSeq;
WinPostMsg (ChildFrameWindow, WM_PAD, MP1, MP2);
```

Example 1: *Code fragments showing how the new version of Stony Brook Modula-2 handles* MPARAM.

OS/2's buffer into a local buffer (which can be any size). Although each call to *GetChar* returns only a single character, it also results in OS/2's (too small) buffer getting transferred to a local buffer (the size of which can be specified by the programmer).

Stony Brook v2.1 should be shipping by the time you read this, and will have many improvements (including a fixed CommPort module). Unfortunately, Stony Brook has changed the use of the OS/2-PM message parameter, *MPARAM*. According to the Microsoft documentation, and the earlier versions of the Stony Brook documentation, *MPARAM* is a 32-bit address. Stony Brook v2.1 has changed the representation of *MPARAM* to a variant record. Figure 1 shows the *record* definition.

MPARAM (a 32-bit address used as a parameter during message passing) is often made up of an *unsigned short* or a *char*. Microsoft provides various macros, such as MPFROM2SHORT, which make the necessary conversions. The original version of Stony Brook provided similar macros. The new version, however, does not use the macros, but instead fills in the variant records with information before passing the parameters. Example 1 provides a code fragment that demonstrates this.

Other Modules

FILES.MOD (Listing Thirteen, page 116) provides access to disk files: opens, reads, writes, and closes. FILES is essentially unchanged from the DOS version except that file name clashes now result in automatic renaming of the file, rather than asking the user whether or not to overwrite (for example, if a duplicate file name of FILENAME.TYP is received, it is renamed to FILENAME.K0).

KH.DEF (Listing Fourteen, page 118), KH.MOD (Listing Fifteen, page 118), and PCKERMIT.H (Listing Sixteen, page 118) contain definitions of constants (e.g., *IDM_KERMIT*) that are used by the resources. Each menu item, radio button, icon, and so on, has its own unique identifier. The Modula-2 program gets this information from KH.DEF (KH.MOD is empty). The resources themselves get the information from PC-

```
TYPE
   MPARAM = RECORD
      CASE : INTEGER OF
         1   :  B1, B2, B3, B4 : BYTE;
         2   :  W1, W2 : WORD;
         3   :  L : LONGINT;
      END;
   END;
```

Figure 1: *Record type definition for* MPARAM.

KERMIT.H. In a C program, both the program and the resources would get the information from the (.H) header file; but because the Modula-2 compiler cannot read .H files, the information must be duplicated in a .DEF file.

PAD.MOD is the packet assembler/disassembler. Because of its length, the PAD.MOD listing is not included in this article (but it is available directly from *DDJ*). Except for the screen output (via message passing) to keep the user informed, and multiple file sending, this module is essentially unchanged from the original DOS version (see "Kermit Meets Modula-2," *DDJ*, May 1989). During Kermit file transfer, PAD assembles the data into packets and passes the packets on to the DataLink module.

The changes involve the OS/2-PM message passing facility, *WinPOstMsg( )* — which replaces the ordinary terminal output for progress messages, and collecting all of the output routines into a single function. This rather strange organization is necessary because most of the PAD module runs in a non-message-queue thread, which cannot perform I/O. The function that collects the output routines is called from the main thread, which has a message queue.

DATALINK.MOD gets the packets from PAD, adds SOH and CheckSum, and transmits them (as well as the reverse). Again, this module is virtually unchanged from the DOS version.

PCKERMIT.RC (Listing Eighteen, page 119) is the resource file that contains all resources except bitmaps. These resources include menus, dialog boxes, and keyboard accelerators. Bitmaps, which include icons, fonts, and graphics bit images, are in separate files. A bit-image file called PCKERMIT.ICO (see "Availability" on page 3) contains the icon (a picture of Kermit the Frog) that is displayed when the program is minimized. Icons are created interactively using a program called ICONEDIT, which is essentially a drawing program.

Although I created all of the dialog boxes using the Microsoft's DLGBOX program to separately create the resources for the dialog boxes, I then merged all of the code together into PCKERMIT.RC. I added the menus and accelerator tables manually. The resource compiler converts the .RC file into a .RES file, which is then linked into the .EXE file.

PCKERMIT.EDF (Listing Nineteen, page 121) is the .EXE definition file; it contains information for the linker (e.g., stack size, window procedures exported, procedures imported via DLLs, and mode). In a C project, this file would be called PCKERMIT.DEF.

Conclusion

While it is a steep learning curve to become familiar with the OS/2 Kernel API and the OS/2 Presentation Manager API, it is well worth the climb. Once you have a working skeleton, adding the menus, keyboard accelerators, message boxes, and dialog boxes is really quite easy. Having these things out of the way allows you to concentrate upon the application itself. Except for console I/O, which is heavily influenced by the Presentation Manager, much of any OS/2-PM application is the same as in a more traditional environment.

This article does not address the *Gpi* functions, and I suspect that will involve another steep climb up a slippery slope. Again, I feel the view will be worth the climb — learning to use the PM functions is certainly easier than writing similar functionality into your own code.

Finally, I would like to mention that since this article was written I have continued to enhance PCKermit. The enhanced version is available directly from me at the address given at the beginning of this article.

DDJ

(Listings begin on page 108.)

Vote for your favorite feature/article.
Circle Reader Service **No. 7.**

Listing One *(Text begins on page 16.)*

```
;*********************************************************************
;* PROT - A 386 protected mode DOS extender                          *
;* Copyright (C) 1989, by Al Williams  All rights reserved.          *
;* Permission is granted for non-commercial use of this software.    *
;* You are expressly prohibited from selling this software,          *
;* distributing it with another product, or removing this notice.    *
;* If you distribute this software to others in any form, you must   *
;* distribute all of the files that are listed below:                *
;* PROT.ASM     - The main routines and protected mode support.      *
;* EQUMAC.INC   - Equates and macros.                                *
;* STACKS.INC   - Stack segments.                                    *
;* GDT.INC      - Global descriptor table.                           *
;* INT386.INC   - Protected mode interrupt handlers.                 *
;* PMDEMO.PM    - Example user code.                                 *
;* PMPWD.PM     - Alternate example code.                            *
;* FBROWSE.PM   - Complete sample application.                       *
;* TSS.INC      - Task state segments.                               *
;* CODE16.INC   - 16 bit DOS code (entry/exit).                      *
;* PMASM.BAT    - MASM driver for assembling PROT programs.          *
;* PTASM.BAT    - TASM driver for assembling PROT programs.          *
;* To assemble:    MASM /DPROGRAM=pname PROT.ASM,,PROT.LST;          *
;* To link:        LINK PROT;                                        *
;*      pname is the program name (code in pname.PM)                 *
;*      if pname is ommited, USER.PM is used                         *
;* The resulting .EXE file is executable from the DOS prompt.        *
;* This file is: PROT.ASM, the main protected mode code.             *
;*********************************************************************

.XLIST
.LALL

.386P

; Name program if PROGRAM is defined
IFDEF       PROGRAM
VTITLE      MACRO   PNAME       ; temporary macro to title program
            TITLE   PNAME
            ENDM
            VTITLE  %PROGRAM
            PURGE   VTITLE      ; delete macro
; equates and macros
INCLUDE     EQUMAC.INC

; stack segments
INCLUDE     STACKS.INC

; Global descriptor table definitons
INCLUDE     GDT.INC

; interrupt code
INCLUDE     INT386.INC

; this is required to find out how large PROT is
ZZZGROUP    GROUP   ZZZSEG

;*********************************************************************
; 32 bit data segment
DAT32       SEGMENT PARA PUBLIC 'DATA32' USE32
DAT32BEG    EQU     $

; 32 bit stack values
SLOAD       DD      OFFSET SSEG321-1
SSLD        DW      SEL_STACK

; This location will hold the address for the PMODE IDT
NEWIDT      EQU     THIS FWORD
            DW      (IDTEND-IDTBEG)-1
IDTB        DD      0           ; filled in at runtime

; PSP segment address
_PSP        DW      0

; video variables for the OUCH and related routines
CURSOR      DD      0           ; cursor location
COLOR       DB      7           ; display cursor

; temp vars for some non reentrant interrupt routines
STO1        DD      0
STO2        DD      0
STO3        DD      0
STO4        DD      0
SAV_DS      DD      0
SAV_ES      DD      0
SAV_GS      DD      0
SAV_FS      DD      0

BPON        DB      0           ; Enables conditional breakpoints

; Debug Dump variables
DUMP_SEG    DW      0           ; if zero don't dump memory
DUMP_OFF    DD      0           ; Offset to start at
DUMP_CNT    DD      0           ; # of bytes to dump

; Break & critical error handler variables
BREAKKEY    DB      0           ; break key occurred
CRITICAL    DB      0           ; critical error occured
CRITAX      DW      0           ; critical error ax
CRITDI      DW      0           ; critical error di
CRITBP      DW      0           ; critical error bp
CRITSI      DW      0           ; critical error si

; Address of user's break handler
BREAK_HANDLE EQU    THIS FWORD
BRK_OFF     DD      0
BRK_SEG     DW      0

; Address of user's critical error handler
CRIT_HANDLE EQU     THIS FWORD
CRIT_OFF    DD      OFFSET DEF_CRIT
CRIT_SEG    DW      SEL_CODE32

; Message for default critical error handler
CRITMSG     DB      'A critical error has occured.',13,10
            DB      '<A>bort, <R>etry, <F>ail? $'

; here is where vm86 int's stack up pl0 esp's
INTSP       DD      $+PVSTACK+4
            DB      PVSTACK DUP (0)

; Default VM86CALL parameter block
PINTFRAME   VM86BLK <>

; interface block for critical error handler
CINTFRAME   VM86BLK <>

; hardware interrupt vm86 block
HINTFRAME   VM86BLK <>

; storage for the original PIC interrupt mask registers
INTMASK     DB      0
INTMASKAT   DB      0

DAT32END    EQU     $
DAT32       ENDS

;*********************************************************************
; Begin 32 bit code segment

SEG32       SEGMENT PARA PUBLIC 'CODE32' USE32
            ASSUME  CS:SEG32, DS:DAT32
PCODE       PROC
SEG32BEG    EQU     $

; Start of protected mode code. We jump here from inside CODE16.INC
SEG32ENT:   MOV     AX,SEL_DATA  ; 1st order of business:
            MOV     DS,AX        ; load up segment registers
            LSS     ESP, FWORD PTR SLOAD
            MOV     AX,SEL_VIDEO
            MOV     ES,AX
            MOV     AX,SEL_DATA0
            MOV     FS,AX
            MOV     AX,SEL_GDT
            MOV     GS,AX
; set up IDT
            CALL32S MAKIDT
; reprogram pic(s)
            IN      AL,21H
            MOV     INTMASK,AL
IF          ATCLASS
            IN      AL,0A1H
            MOV     INTMASKAT,AL
            MOV     AL,11H
            OUT     0A0H,AL
            OUT     20H,AL
            IDELAY
            MOV     AL,28H
            OUT     0A1H,AL
            MOV     AL,20H
            OUT     21H,AL
            IDELAY
            MOV     AL,2
            OUT     0A1H,AL
            MOV     AL,4
            OUT     21H,AL
            IDELAY
            MOV     AL,1
            OUT     0A1H,AL
            OUT     21H,AL
            IDELAY
            MOV     AL,INTMASKAT
            OUT     0A1H,AL
            MOV     AL,INTMASK
            OUT     21H,AL
ELSE
; INBOARD PC Code
            MOV     AL,13H
            OUT     20H,AL
            MOV     AL,20H
            OUT     21H,AL
            MOV     AL,9
            OUT     21H,AL
            MOV     AL,INTMASK
            OUT     21H,AL
ENDIF
            STI                 ; enable interrupts

; *** Start user code with TSS (req'd for vm86 op's etc.)
            MOV     AX,TSS0
            LTR     AX
            JMPABS32 TSS1,0
PCODE       ENDP

;*** 32 bit support routines
; This routine creates the required IDT. This is only a subroutine to keep
; from cluttering up the main code, since you aren't likely to call it again.
; Assumes that all ISR routines are of fixed length and in sequence. After
; makidt has built the table, you can still replace individual INT gates with
; your own gates (see make_gate)
MAKIDT      PROC    NEAR
            PUSH    ES
            MOV     AX,IDTABLE
            MOVZX   EAX,AX
            SHL     EAX,4
            ADD     EAX,OFFSET IDTBEG
            MOV     IDTB,EAX
            MOV     AX,SEL_IDT
            MOV     ES,AX
            XOR     AL,AL
```

(continued on page 82)

Listing One *(Listing continued, text begins on page 16.)*

```
; Make all interrupt gates DPL=3
                MOV     AH,INTR_GATE OR DPL3
                MOV     CX,SEL_ICODE
                MOV     EDX,OFFSET IDTBEG
                XOR     SI,SI
                MOV     EBX,OFFSET INT0
IDTLOOP:        CALL32F SEL_CODE32,MAKE_GATE
                ADD     EBX,INT1-INT0
                ADD     SI,8
; loop form max # of interrupts
                CMP     SI,(TOPINT+1)*8
                JB      SHORT IDTLOOP
                LIDT    NEWIDT
                POP     ES
                RET
MAKIDT          ENDP

; This routine is just like the real mode make_desc
; EBX=base  ECX=limit   AH=ARB  AL=0 or 1 for 16 or 32 bit
; SI=selector (TI&RPL ignored) and ES:EDX is the table base address
MAKE_SEG        PROC    FAR
                PUSH    ESI
                PUSH    EAX
                PUSH    ECX
                MOVZX   ESI,SI
                SHR     SI,3            ; adjust to slot #
                SHL     AL,6            ; shift size to right bit position
                CMP     ECX,0FFFFFH     ; see if you need to set G bit
                JLE     OKLIM
                SHR     ECX,12          ; div by 4096
                OR      AL,80H          ; set G bit
OKLIM:          MOV     ES:[EDX+ESI*8],CX
                SHR     ECX,16
                OR      CL,AL
                MOV     ES:[EDX+ESI*8+6],CL
                MOV     ES:[EDX+ESI*8+2],BX
                SHR     EBX,16
                MOV     ES:[EDX+ESI*8+4],BL
                MOV     ES:[EDX+ESI*8+5],AH
                MOV     ES:[EDX+ESI*8+7],BH
                POP     ECX
                POP     EAX
                POP     ESI
                RET
MAKE_SEG        ENDP

; This routine make gates -- AL=WC if applicable -- AH=ARB  -- EBX=offset
; CX=selector  -- ES:EDX=table base  --  SI= selector (TI&RPL ignored)
MAKE_GATE       PROC    FAR
                PUSH    ESI
                PUSH    EBX
                SHR     SI,3
                MOVZX   ESI,SI
                MOV     ES:[EDX+ESI*8],BX
                MOV     ES:[EDX+ESI*8+2],CX
                MOV     ES:[EDX+ESI*8+4],AX
                SHR     EBX,16
                MOV     ES:[EDX+ESI*8+6],BX
                POP     EBX
                POP     ESI
                RET
MAKE_GATE       ENDP

; Routine to call BIOS/DOS. NOT REENTRANT (but so what? DOS isn't either)
CALL86          PROC    FAR
                PUSH    DS
                PUSH    GS
                PUSH    FS
RETRY86:
                PUSHAD
                PUSHFD
                PUSH    ES:[EBX+40]  ; save new ebx
                PUSH    EBX
                PUSH    ES
                INT     30H          ; call PROT
                PUSH    SEL_DATA
                POP     DS
                POP     ES
                XCHG    EBX,[ESP]
                POP     ES:[EBX+40]
                PUSHFD
                CMP     BREAKKEY,0   ; see if break occured
                JZ      SHORT NOBRKCHECK
                CMP     BRK_SEG,0    ; see if user has brk handler
                JZ      SHORT NOBRKCHECK
                                     ; call user's break handler
                MOV     BREAKKEY,0
                CALL    FWORD PTR BREAK_HANDLE
NOBRKCHECK:
                CMP     CRITICAL,0   ; see if critical error
                JZ      SHORT NOCRITCK
                CMP     CRIT_SEG,0   ; see if critical error handler
                JZ      SHORT NOCRITCK
                                     ; call critical error handler
                PUSH    EAX
                XOR     AL,AL
                MOV     CRITICAL,AL
                CALL    FWORD PTR CRIT_HANDLE
                OR      AL,AL        ; AL=0? FAIL
                JNZ     SHORT RETRY?
                POP     EAX
                POPFD
                STC                  ; make sure carry is set
                PUSHFD
                JMP     SHORT NOCRITCK
RETRY?:         DEC     AL           ; AL=1? RETRY
                JNZ     SHORT CABORT
```

(continued on page 84)

Listing One *(Listing continued, text begins on page 16.)*

```
; To retry an error, we set up everything the way it was and
; redo the interrupt. This is cheating (a little), and may not
; work in every possible case, but it seems to work in all the cases tried.
                POP     EAX
                POPFD
                POP     ES:[EBX+40]
                POPFD
                POPAD
                JMP     SHORT RETRY86
CABORT:         POP     EAX             ; ABORT
                POPFD
                LEA     ESP,[ESP+40] ; balance stack
                MOV     AL,7FH          ; DOS error=7FH
                BACK2DOS
NOCRITCK:
                POPFD
                LEA     ESP,[ESP+40] ; balance stack
                PUSHFD
; see if segment save requested
                CMP     BYTE PTR ES:[EBX],0
                JZ      NOSEGS
; load parameter block from static save area
                PUSH    EAX
                MOV     EAX,SAV_FS
                MOV     ES:[EBX+28],EAX
                MOV     EAX,SAV_DS
                MOV     ES:[EBX+24],EAX
                MOV     EAX,SAV_ES
                MOV     ES:[EBX+20],EAX
                MOV     EAX,SAV_GS
                MOV     ES:[EBX+32],EAX
                POP     EAX
NOSEGS:
                POPFD
                POP     FS
                POP     GS
                POP     DS
                MOV     EBX,ES:[EBX+40]
                RET
CALL86          ENDP

; Directly clear page 0 of the screen
CLS             PROC    FAR
                PUSHFD
                PUSH    DS
                PUSH    ES
                PUSH    EDI
                PUSH    ECX
                PUSH    EAX
                MOV     CX,SEL_VIDEO
                MOV     ES,CX
                MOV     CX,SEL_DATA
                MOV     DS,CX
                CLD
                MOV     EDI,0
                MOV     ECX,2000
                MOV     AX,0720H
                REP     STOSW
                XOR     ECX,ECX
                MOV     CURSOR,ECX
                POP     EAX
                POP     ECX
                POP     EDI
                POP     ES
                POP     DS
                POPFD
                RET
CLS             ENDP

; Outputs message to screen -- ASCIIZ pointer in ds:ebx - modifies ebx
MESSOUT         PROC    FAR
                PUSH    EAX
NXT:            MOV     AL,[EBX]
                INC     EBX
                OR      AL,AL
                JNZ     SHORT SKIP
                POP     EAX
                RET
SKIP:           CALL32F SEL_CODE32, OUCH
                JMP     SHORT NXT
MESSOUT         ENDP

; Performs CR/LF sequence to screen using OUCH
CRLF            PROC    FAR
                PUSH    EAX
                MOV     AL,13
                CALL32F SEL_CODE32,OUCH
                MOV     AL,10
                CALL32F SEL_CODE32,OUCH
                POP     EAX
                RET
CRLF            ENDP

; Character and digit output routines
; hexout4 - print longword in EAX in hex
; hexout2 - print word in AX in hex
; hexout  - print byte in AL in hex
; ouch    - print ASCII character in AL
OUTPUT          PROC    FAR
; print longword in eax
HEXOUT4         LABEL   FAR
                PUSH    EAX
                SHR     EAX,16
                CALL32F SEL_CODE32,HEXOUT2
                POP     EAX
; print word in ax
HEXOUT2         LABEL   FAR
                PUSH    EAX
                MOV     AL,AH
                CALL32F SEL_CODE32, HEXOUT
                POP     EAX
; print a hex byte in al
HEXOUT          LABEL   FAR
                MOV     BL,AL
                AND     AX,0F0H
                SHL     AX,4
                MOV     AL,BL
                AND     AL,0FH
                ADD     AX,'00'
                MOV     BL,AL
                MOV     AL,AH
                CALL32F SEL_CODE32, HEX1DIG
                MOV     AL,BL
HEX1DIG:        CMP     AL,'9'
                JBE     SHORT H1DIG
                ADD     AL,'A'-'0'-0AH
H1DIG:
OUCH            LABEL   FAR
                PUSH    EDI
                PUSH    EAX
                PUSH    DS
                PUSH    ES
                PUSH    ECX
                MOV     CX,SEL_VIDEO
                MOV     ES,CX
                MOV     CX,SEL_DATA
                MOV     DS,CX
                POP     ECX
                MOV     AH,COLOR
                MOV     EDI,CURSOR
                CMP     EDI,2000        ; rolling off the screen?
                JB      NOSCROLL
; scroll screen if required
                PUSH    DS
                PUSH    ES
                POP     DS
                PUSH    ESI
                PUSH    ECX
                PUSH    EDI
                CLD
                MOV     ECX,960
                XOR     EDI,EDI
                MOV     ESI,160
                REP     MOVSD
                POP     EDI
                SUB     EDI,80
                POP     ECX
                POP     ESI
                POP     DS
NOSCROLL:       CMP     AL,0DH
                JZ      SHORT CR
                CMP     AL,0AH
                JZ      SHORT LF
; write to screen
                MOV     ES:[EDI*2],AX
                INC     EDI
                JMP     SHORT OUCHD
CR:             PUSH    EDX
                PUSH    ECX
                MOV     EAX,EDI
                XOR     EDX,EDX
                MOV     ECX,80
                DIV     ECX
                SUB     EDI,EDX
                POP     ECX
                POP     EDX
                JMP     SHORT OUCHD
LF:             ADD     EDI,50H
OUCHD:          MOV     CURSOR,EDI      ; update cursor
                POP     ES
                POP     DS
                POP     EAX
                POP     EDI
                RET
OUTPUT          ENDP
; Default critical error handler
DEF_CRIT        PROC    FAR
                PUSH    ES
                PUSH    EBX
                PUSH    EDX
                MOV     BX,SEL_DATA
                MOV     ES,BX
                ASSUME  DS:NOTHING, ES:DAT32
; load critical error handler's private stack
                MOV     BX,CSTACK
                MOV     CINTFRAME.VMSS,EBX
                MOV     EBX,OFFSET CSTACK
                MOV     CINTFRAME.VMESP,EBX
                MOV     BX,DAT32
                MOV     CINTFRAME.VMDS,EBX
                MOV     BX,21H
                MOV     CINTFRAME.VMINT,EBX
                MOV     EBX, OFFSET CINTFRAME
                MOV     EDX,OFFSET CRITMSG
                MOV     AH,9
                PUSH    EBX
                VM86CALL                ; print message
                POP     EBX
CLOOP:
                MOV     AH,7
                PUSH    EBX
                VM86CALL                ; get keystroke
                POP     EBX
; ignore function keys
                OR      AL,AL
                JZ      SHORT CRITFNKEY
                MOV     AH,AL
                OR      AL,20H          ; convert to lower case
                CMP     AL,'a'
```

(continued on page 86)

Listing One *(Listing continued, text begins on page 16.)*

```
                JNZ     SHORT CFAIL?
                MOV     AL,2
                JMP     SHORT CREXIT
CFAIL?:         CMP     AL,'f'
                JNZ     SHORT CRETRY?
                XOR     AL,AL
                JMP     SHORT CREXIT
CRETRY?:
                CMP     AL,'r'
                MOV     AL,1
                JNZ     SHORT CRITBAD
CREXIT:         MOV     DL,AH           ; echo letter + CRLF
                MOV     AH,2
                PUSH    EAX
                PUSH    EBX
                VM86CALL
                POP     EBX
                MOV     AH,2
                MOV     DL,0DH
                PUSH    EBX
                VM86CALL
                POP     EBX
                MOV     AH,2
                MOV     DL,0AH
                VM86CALL
                POP     EAX
                POP     EDX
                POP     EBX
                POP     ES
                RET
CRITFNKEY:
                MOV     AH,7
                PUSH    EBX
                VM86CALL                ; ignore fn key/alt-key
                POP     EBX
CRITBAD:
                MOV     DL,7
                MOV     AH,2
                PUSH    EBX
                VM86CALL                ; unknown input - ring bell
                POP     EBX
                JMP     SHORT CLOOP
DEF_CRIT        ENDP

SEG32END        EQU     $
SEG32           ENDS

;*****************************************************************
; user program - PROT includes the file defined by the variable PROGRAM.
; convoluted method to make MASM take a string equate for an include filename

TEMPINCLUDE     MACRO   FN              ; ; temporary macro
                INCLUDE &FN&.PM
                ENDM
TEMPINCLUDE     %PROGRAM

PURGE           TEMPINCLUDE             ; delete macro

; task state segments
INCLUDE         TSS.INC

; 16 bit code (DOS entry/exit)
INCLUDE         CODE16.INC
; Segment to determine the last memory address
ZZZSEG          SEGMENT PARA PUBLIC 'ZZZ' USE16
ZZZSEG          ENDS
ELSE
IF2
                %OUT    You must specify a program title
                %OUT    use: MASM /DPROGRAM=PNAME PROT.ASM...
ENDIF
                .ERR
ENDIF
                END     ENTRY
```

End Listing One

Listing Two

```
;*****************************************************************
;* PROT - A 386 protected mode DOS extender                      *
;* Copyright (C) 1989, by Al Williams -- All rights reserved.    *
;* Permission is granted for non-commercial use of this software *
;* subject to certain conditions (see PROT.ASM).                 *
;* This file is: GDT.INC, the Global Descriptor Table definitions. *
;*****************************************************************
; See EQUMAC.INC for an explanation of the DESC macro
GDTSEG          SEGMENT PARA PUBLIC 'CODE32' USE32
GDT             EQU     $               ; GDT space
                DESC    SEL_NULL        ; DUMMY NULL SELECTOR
                DESC    SEL_CODE16      ; 16 BIT CODE SEGMENT
                DESC    SEL_DATA0       ; 4GB SEGMENT
                DESC    SEL_CODE32      ; 32 BIT CODE SEGMENT
                DESC    SEL_STACK       ; 32 BIT STACK
                DESC    SEL_RDATA       ; REAL MODE LIKE DATA SEG
                DESC    SEL_GDT         ; GDT ALIAS
                DESC    SEL_VIDEO       ; VIDEO MEMORY
                DESC    SEL_DATA        ; 32 BIT DATA
                DESC    SEL_IDT         ; IDT ALIAS
                DESC    SEL_ICODE       ; ISR SEGMENT
                DESC    SEL_TSS0        ; DUMMY TASK BLOCK
                DESC    TSS0            ; SAME (MUST FOLLOW SEL_TSS0)
                DESC    SEL_TSS1        ; MAIN TASK BLOCK
                DESC    TSS1            ; SAME (MUST FOLLOW SEL_TSS1)
                DESC    SEL_UCODE       ; USER CODE
                DESC    SEL_UDATA       ; USER DATA
                DESC    SEL_PSP         ; DOS PSP
                DESC    SEL_FREE        ; FREE DOS MEMORY
                DESC    SEL_EXT         ; EXTENDED MEMORY
                DESC    SEL_ENV         ; ENVIROMENT
GDTEND          =       $
GDTSEG          ENDS
```

End Listing Two

Listing Three

```
;*****************************************************************
;* PROT - A 386 protected mode DOS extender                      *
;* Copyright (C) 1989, by Al Williams -- All rights reserved.    *
;* Permission is granted for non-commercial use of this software *
;* subject to certain conditions (see PROT.ASM).                 *
;* This file is: EQUMAC.INC, assorted macros and equates.        *
;*****************************************************************
; EQUates the user may wish to change
ATCLASS         EQU     1               ; 1=AT/386  0=INBOARD 386/PC
DOSSTACK        EQU     200H            ; stack size for DOS startup
VM86STACK       EQU     200H            ; stack size for VM86 int calls
CRITSTACK       EQU     30H             ; stack size for crit err handler
PMSTACK         EQU     1000H           ; stack size for p-mode stack
PVSTACK         EQU     260             ; p10/vm86 psuedo stack size
; Maximum protected mode interrupt # defined
                TOPINT  EQU 30H
; The critical error handler works different for DOS 2.X than for other DOS
; versions. In 99% of the cases it won't make any difference if you compile
; with DOS=2.... major dos version number (2, 3 or 4)
                DOS     EQU 3

; parameter block to interface for int 30H (call86 & VM86CALL)
VM86BLK         STRUC
VMSEGFLAG       DD      0               ; restore segment registers (flag)
VMINT           DD      0               ; interrupt number
VMFLAGS         DD      0               ; EFLAGS
VMESP           DD      0               ; ESP
VMSS            DD      0               ; SS
VMES            DD      0               ; ES
VMDS            DD      0               ; DS
VMFS            DD      0               ; FS
VMGS            DD      0               ; GS
VMEBP           DD      0               ; EBP
VMEBX           DD      0               ; EBX
VM86BLK         ENDS

; Access rights equates. Use these with make_desc or make_seg
RO_DATA         EQU     90H             ; r/o data
RW_DATA         EQU     92H             ; r/w data
RO_STK          EQU     94H             ; r/o stack
RW_STK          EQU     96H             ; r/w stack
EX_CODE         EQU     98H             ; exec only code
ER_CODE         EQU     9AH             ; read/exec code
CN_CODE         EQU     9CH             ; exec only conforming code
CR_CODE         EQU     9EH             ; read/exec conforming code
LDT_DESC        EQU     82H             ; LDT entry
TSS_DESC        EQU     89H             ; TSS entry
; use these with make_gate
CALL_GATE       EQU     8CH             ; call gate
TRAP_GATE       EQU     8FH             ; trap gate
INTR_GATE       EQU     8EH             ; int gate
TASK_GATE       EQU     85H             ; task gate

; dpl equates
DPL0            EQU     0
DPL1            EQU     20H
DPL2            EQU     40H
DPL3            EQU     60H

; macro definitons

; other macros use this to error check parameters
; Give an error if last is blank or toomany is not blank
ERRCHK          MACRO   LAST,TOOMANY
IFNB            <TOOMANY>
IF2
                %OUT    Too many parameters
ENDIF
                .ERR
ENDIF
IFB             <LAST>
IF2
                %OUT    Not enough parameters
ENDIF
                .ERR
ENDIF
                ENDM

; Perform absolute 16 bit jump (in a 16 bit segment)
JMPABS          MACRO   A,B,ERRCK
                ERRCHK  B,ERRCK
                DB      0EAH            ; ; absoulte 16 bit jump
                DW      OFFSET B
```

(continued on page 88)

Listing Three *(Listing continued, text begins on page 16.)*

```
            DW      A
            ENDM

; Peform absolute 32 bit jump (in a 32 bit segment)
JMPABS32    MACRO   A,B,ERRCK
            ERRCHK  B,ERRCK
            DB      0EAH        ; ; absolute 32 bit jump
            DD      OFFSET B
            DW      A
            ENDM
; this generates a correct 32 bit offset for a proc call
; since MASM doesn't sign extend 32 bit relative items
CALL32S     MACRO   LBL,ERRCK   ; ; short call
            ERRCHK  LBL,ERRCK
            DB      0E8H
            DD      LBL-($+4)
            ENDM

CALL32F     MACRO   SG,LBL,ERRCK ; ; far call
            ERRCHK  LBL,ERRCK
            DB      9AH
            DD      OFFSET LBL
            DW      SG
            ENDM

JMP32S      MACRO   LBL,ERRCK   ; ; short jump
            ERRCHK  LBL,ERRCK
            DB      0E9H
            DD      LBL-($+4)
            ENDM
; conditional jump macro
; jcc32 uses condition codes used in Intel literature
JCC32       MACRO   CONDX,LBL,ERRCK
            ERRCHK  LBL,ERRCK
            DB      0FH
IFIDNI      <CONDX>,<A>
            DB      87H
ELSEIFIDNI  <CONDX>,<NBE>
            DB      87H
ELSEIFIDNI  <CONDX>, <AE>
            DB      83H
ELSEIFIDNI  <CONDX>, <C>
            DB      82H
ELSEIFIDNI  <CONDX>, <NAE>
            DB      82H
ELSEIFIDNI  <CONDX>, <B>
            DB      82H
ELSEIFIDNI  <CONDX>, <BE>
            DB      86H
ELSEIFIDNI  <CONDX>, <E>
            DB      84H
ELSEIFIDNI  <CONDX>, <Z>
            DB      84H
ELSEIFIDNI  <CONDX>, <G>
            DB      8FH
ELSEIFIDNI  <CONDX>, <GE>
            DB      8DH
ELSEIFIDNI  <CONDX>, <L>
            DB      8CH
ELSEIFIDNI  <CONDX>, <LE>
            DB      8EH
ELSEIFIDNI  <CONDX>, <NA>
            DB      86H
ELSEIFIDNI  <CONDX>, <NB>
            DB      83H
ELSEIFIDNI  <CONDX>, <NC>
            DB      83H
ELSEIFIDNI  <CONDX>, <NGE>
            DB      8CH
ELSEIFIDNI  <CONDX>, <NL>
            DB      8DH
ELSEIFIDNI  <CONDX>, <NO>
            DB      81H
ELSEIFIDNI  <CONDX>, <NP>
            DB      8BH
ELSEIFIDNI  <CONDX>, <NS>
            DB      89H
ELSEIFIDNI  <CONDX>, <NZ>
            DB      85H
ELSEIFIDNI  <CONDX>, <O>
            DB      80H
ELSEIFIDNI  <CONDX>, <P>
            DB      8AH
ELSEIFIDNI  <CONDX>, <PE>
            DB      8AH
ELSEIFIDNI  <CONDX>, <PO>
            DB      8BH
ELSEIFIDNI  <CONDX>, <S>
            DB      88H
ELSE
            %OUT    JCC32: Unknown condition code
            .ERR
ENDIF
            DD      LBL-($+4)
            ENDM

; Override default operand size
OPSIZ       MACRO   NOPARM      ; ; op size overide
            ERRCHK  X,NOPARM
            DB      66H
            ENDM
; Override default address size
ADSIZ       MACRO   NOPARM      ; ; address size overide
            ERRCHK  X,NOPARM
```

(continued on page 90)

Listing Three *(Listing continued, text begins on page 16.)*

```
                DB      67H
                ENDM
; delay macro for interrupt controller access
IDELAY          MACRO   NOPARM
                LOCAL   DELAY1,DELAY2
                ERRCHK  X,NOPARM
                JMP     SHORT DELAY1
DELAY1:         JMP     SHORT DELAY2
DELAY2:
                ENDM

; BREAKPOINT MACROS

; MACRO to turn on NBREAKPOINTS. If used with no arguments (or a 1), this
; macro makes NBREAKPOINT active if used with an argument > 1, NBREAKPOINT
; will break after that many passes
BREAKON         MACRO   ARG,ERRCK
                ERRCHK  X,ERRCK
                PUSH    DS
                PUSH    SEL_DATA
                POP     DS
                PUSH    EAX
                IFB     <ARG>
                MOV     AL,1
                ELSE
                MOV     AL,&ARG
                ENDIF
                MOV     BPON,AL
                POP     EAX
                POP     DS
                ENDM
; Turns off NBREAKPOINT
BREAKOFF        MACRO   NOPARAM
                ERRCHK  X,NOPARAM
                PUSH    DS
                PUSH    SEL_DATA
                POP     DS
                PUSH    EAX
                XOR     AL,AL
                MOV     BPON,AL
                POP     EAX
                POP     DS
                ENDM
BREAKPOINT      MACRO   NOPARM
                ERRCHK  X,NOPARM
                INT     3
                ENDM
; Counter breakpoint - use BREAKON to set count control

; BREAKPOINT with memory dump.
; usage: BREAKDUMP seg_selector, offset, number_of_words
BREAKDUMP       MACRO   SEG,OFF,CNT,ERRCK
                ERRCHK  CNT,ERRCK
                PUSH    EAX
                PUSH    DS
                MOV     AX,SEL_DATA
                MOV     DS,AX
                MOV     AX,&SEG
                MOV     DUMP_SEG,AX
                MOV     EAX,OFFSET &OFF
                MOV     DUMP_OFF,EAX
                MOV     EAX,&CNT
                MOV     DUMP_CNT,EAX
                POP     DS
                POP     EAX
                BREAKPOINT
                ENDM
NBREAKDUMP      MACRO   SEG,OFF,CNT,ERRCK
                ERRCHK  CNT,ERRCK
                LOCAL   NONBP
                PUSH    DS
                PUSH    SEL_DATA
                POP     DS
                PUSHFD
                OR      BPON,0
                JZ      NONBP
                DEC     BPON
                JNZ     NONBP
                POPFD
                POP     DS
                BREAKDUMP SEG,OFF,CNT
NONBP:
                POPFD
                POP     DS
                ENDM

; determine linear address of first free byte of memory (to nearest paragraph)
LOADFREE        MACRO   REG,ERRCK
                ERRCHK  REG,ERRCK
                XOR     E&REG,E&REG
                MOV     &REG,SEG ZZZGROUP
                SHL     E&REG,4
                ENDM

; Set up PINTFRAME  (uses eax). Loads vmstack & vmdata to the ss:esp and
; ds slots in pintframe  -- default ss:esp=ssint1 -- default ds=userdata
PROT_STARTUP MACRO      VMSTACK,VMDATA,ERRCK
                ERRCHK  X,ERRCK
IFB             <VMSTACK>
                MOV     AX,SEG SSINT1
ELSE
                MOV     AX,SEG VMSTACK
ENDIF
                MOV     PINTFRAME.VMSS,EAX
IFB             <VMSTACK>
                MOV     EAX, OFFSET SSINT1
ELSE
                MOV     EAX, OFFSET VMSTACK
```

(continued on page 92)

Listing Three *(Listing continued, text begins on page 16.)*

```
ENDIF
                MOV     PINTFRAME.VMESP,EAX
IFB             <VMDATA>
                MOV     AX,SEG USERDATA
ELSE
                MOV     AX,SEG VMDATA
ENDIF
                MOV     PINTFRAME.VMDS,EAX
                ENDM

; start PROT user segments
PROT_CODE       MACRO   NOPARM
                ERRCHK  X,NOPARM
USERCODE        SEGMENT PARA PUBLIC 'CODE32' USE32
USERCODEBEG     EQU     $
                ASSUME  CS:USERCODE, DS:USERDATA, ES:DAT32
                ENDM

PROT_DATA       MACRO   NOPARM
                ERRCHK  X,NOPARM
USERDATA        SEGMENT PARA PUBLIC 'DATA32' USE32
USERDATABEG     EQU     $
                ENDM

PROT_CODE_END MACRO     NOPARM
                ERRCHK  X,NOPARM
USERCODEEND     EQU     $
USERCODE        ENDS
                ENDM

PROT_DATA_END MACRO     NOPARM
                ERRCHK  X,NOPARM
USERDATAEND     EQU     $
USERDATA        ENDS
                ENDM

; Simplfy programs with no data segment
NODATA          MACRO   NOPARM
                ERRCHK  X,NOPARM
                PROT_DATA
                PROT_DATA_END
                ENDM

; Mnemonic for call86 call
VM86CALL        MACRO   NOPARM
                ERRCHK  X,NOPARM
                CALL32F SEL_CODE32,CALL86
                ENDM

; Mnemonic for dos return
BACK2DOS        MACRO   RC,ERRCK
                ERRCHK  X,ERRCK
IFNB            <RC>
                MOV     AL,RC
ENDIF
                JMPABS32 SEL_CODE16,BACK16
                ENDM
; Variables and macro to create GDT/LDT/IDT entries
C_GDT           =       0
C_LDT           =       0
C_IDT           =       0

; create "next" descriptor with name in table. If no table specified, use GDT
DESC            MACRO   NAME,TABLE,ERRCK
                DQ      0
IFB             <TABLE>
                NAME    = C_GDT
C_GDT           =       C_GDT+8
ELSE
IFIDNI          <TABLE>,<LDT>
; For LDT selectors, set the TI bit to one
                NAME    = C_&TABLE OR 4
ELSE
                NAME    = C_&TABLE
ENDIF
C_&TABLE        =       C_&TABLE+8
ENDIF
                ENDM
```

End Listings

Listing One *(Text begins on page 28.)*

```
; DEVHLP.ASM
; to produce DEVHLP.SYS ("OS/2 Device Driver in a Can")
; Andrew Schulman, 32 Andrew St., Cambridge MA 02139
; with revisions by Art Rothstein, Morgan Labs, San Francisco, CA

; DEF file:
; LIBRARY DEVHLP
; DESCRIPTION 'DEVHLP.SYS (c) Andrew Schulman 1990'
; PROTMODE

; masm devhlp; && link devhlp,devhlp.sys,,,devhlp.def

; put in OS/2 config.sys:
; device=devhlp.sys

; access with DosOpen ("DEVHLPXX"), DosDevIOCTL (category 128, func 60h)

.286p

; the only specific DevHlp that DEVHLP.SYS knows about
VerifyAccess        equ 27h

ioctlpkt struc
                    db 13 dup (?)   ; header
        cat         db ?            ; category
        fun         db ?            ; function
        param       dd ?            ; param area
        dataptr     dd ?            ; data area
ioctlpkt ends

regs    struc
        regs_ax     dw ?
        regs_bx     dw ?
        regs_cx     dw ?
        regs_dx     dw ?
        regs_si     dw ?
        regs_di     dw ?
        regs_ds     dw ?
        regs_es     dw ?
        regs_flags  dw ?
regs    ends

regs_size   equ     size regs

dgroup  group   _DATA

_DATA   segment word public 'DATA'

header  dd -1
        dw 8880h
        dw Strat
        dw 0
        db 'DEVHLPXX'
        db 8 dup (0)

DevHlp  dd 0

dispch  dw Init             ; 0 -- Init
        dw 12 dup (Error)   ; 1..12 -- not supported
        dw DevOp            ; 13 -- DevOpen
        dw DevOp            ; 14 -- DevClose
        dw Error            ; 15 -- not supported
        dw GenIOCtl         ; 16 -- DevIOCtl
        dw 10 dup (Error)   ; 17..26 -- not supported

enddata dw 0
_DATA   ends

_TEXT   segment word public 'CODE'

        assume cs:_TEXT, ds:DGROUP, es:NOTHING

Strat   proc far
        mov di, es:[bx+2]
        and di, 0ffh
        cmp di, 26                      ; max # of commands
        jle Strat1
        call Error
        jmp short Strat2
Strat1: add di, di
        call word ptr [di+dispch]
Strat2: mov word ptr es:[bx+3], ax      ; set request header status
        ret
Strat   endp

; used by DevOpen and DevClose
DevOp   proc near
        mov ax, 0100h
        ret
DevOp   endp

GenIOCtl    proc near
        push es
        push bx
        cmp es:[bx].cat, 128
        jne bad
        cmp es:[bx].fun, 60h
        jne bad
        call Do_DevHlp
        jc bad
        mov ax, 0100h           ; no error
        jmp short done
bad:    mov ax, 8101h           ; error
done:   pop bx
        pop es
        ret
GenIOCtl endp

Do_DevHlp proc near
        ; verify user's access:
        ; VerifyAccess will shut down user's app in the event of error
        mov ax, word ptr es:[bx+17]     ; selector of parameter block
        mov di, word ptr es:[bx+15]     ; offset
        mov cx, regs_size               ; length to be read
        mov dx, VerifyAccess            ; read
        call DevHlp
        jnc ok1
        ret

ok1:    mov ax, word ptr es:[bx+21]     ; selector of data buffer
        mov di, word ptr es:[bx+19]     ; offset
        mov cx, regs_size               ; length to be written
        mov dx, (1 SHL 8) + VerifyAccess ; read/write
        call DevHlp
        jnc ok2
        ret

ok2:    push ds                 ; see if we should verify ds
        lds di, es:[bx].param
        mov ax, [di].regs_ds
        pop ds
        test ax, ax             ; need to verify?
        je nods                 ; skip if no
        xor di, di              ; verify seg:0 for read, 1 byte
        mov cx, 1               ; length
        mov dx, VerifyAccess    ; read=0
        call DevHlp
        jc fini                 ; if carry flag set

nods:   push ds                 ; see if we should verify es
        lds di, es:[bx].param
        mov ax, [di].regs_es
        pop ds
        test ax, ax             ; need to verify?
        je noes                 ; skip if no
        xor di, di              ; verify seg:0 for read, 1 byte
        mov cx, 1               ; length
        mov dx, VerifyAccess    ; read=0
        call DevHlp
        jc fini                 ; if carry flag set

noes:   push ds                 ; going to be bashed!
        push es
        push bx

        ; save DevHlp address on stack so we can change ds
        push word ptr DevHlp+2
        push word ptr DevHlp

        ; get the parameters for DevHlp from regs
        lds di, es:[bx].param
        mov ax, [di].regs_ax
        mov bx, [di].regs_bx
        mov cx, [di].regs_cx
        mov dx, [di].regs_dx
        mov si, [di].regs_si
        mov es, [di].regs_es
        push    [di].regs_ds
        mov di, [di].regs_di
        pop ds

        ; here it is, the whole point of this exercise!
        mov bp, sp
        call dword ptr [bp]
        pop bp          ; pull DevHlp address off stack
        pop bp          ; without changing carry flag
        jc fini

        ; save ES:BX to put in out-regs: destroys DX
        mov bp, es
        mov dx, bx

        ; get back old DS, ES:BX
        pop bx
        pop es
        pop ds

        ; save FLAGS, SI, DS on stack
        pushf
        push si
        push ds

        ; set up regs to return to the app
        lds si, es:[bx].dataptr
        mov [si].regs_ax, ax
        pop [si].regs_ds
        pop [si].regs_si
        pop [si].regs_flags
        mov [si].regs_cx, cx
        mov [si].regs_bx, dx
        mov [si].regs_es, bp
        mov [si].regs_di, di
        clc
fini:   ret
Do_DevHlp   endp

Error   proc near
        mov ax, 8103h
        ret
Error   endp

Init    proc near
        mov ax, es:[bx+14]
        mov word ptr DevHlp, ax
        mov ax, es:[bx+16]
        mov word ptr DevHlp+2, ax
```

(continued on page 96)

Listing One *(Listing continued, text begins on page 28.)*

```
        mov word ptr es:[bx+14], offset _TEXT:Init       ; end of code
        mov word ptr es:[bx+16], offset DGROUP:enddata
        mov ax, 0100h
        ret
Init    endp

_TEXT   ends

        end
```

End Listing One

Listing Two

```
/* DRVRLIST.C
   list the device drivers in OS/2
   Art Rothstein, 1990

   we assume the first driver in the chain is NUL and is in the global data
   segment, and that the second driver (CON) is the same segment.

   cl -AL drvrlist.c (four-byte data pointers required for memchr)
*/
#define INCL_DOSDEVICES
#include <os2.h>
#include <process.h>
#include <stdio.h>
#include <string.h>
#include "devhlp.h"

USHORT devhlp ;

SEL MakeSel( SEL selValue)
{
  extern USHORT devhlp ;
  REGS regs ;
  USHORT ret ;

  regs.dx = DevHlp_VirtToPhys ;          // function requested
  regs.ds = selValue ;                   // selector
  regs.es = 0 ;                          // avoid trap
  regs.si = 0 ;                          // offset
  ret = DosDevIOCtl( &regs, &regs, 0x60, 128, devhlp) ;
  if ( ret != 0  ||  regs.flags.carry != 0)
    return 0 ;
  // physical address in ax:bx
  regs.cx = 0 ;                                  // limit 65,535
  regs.dx = MAKEUSHORT( DevHlp_PhysToUVirt, UVirt_ReadWrite);
  regs.es = 0 ;                                  // avoid trap
  ret = DosDevIOCtl( &regs, &regs, 0x60, 128, devhlp) ;
  if ( ret != 0  ||  regs.flags.carry != 0)      // if error
    return 0 ;
  return regs.es ;                               // return the selector
}

BOOL ReleaseSel( SEL selValue)
{
  extern USHORT devhlp ;
  REGS regs ;
  USHORT ret ;

  regs.ax = selValue ;                           // selector to free
  regs.dx = MAKEUSHORT( DevHlp_PhysToUVirt, UVirt_Release);
  regs.ds = 0 ;                                  // safety
  regs.es = 0 ;
  ret = DosDevIOCtl( &regs, &regs, 0x60, 128, devhlp) ;
  if ( ret != 0  ||  regs.flags.carry != 0)      // if error
    return FALSE ;
  return TRUE ;                                  // successful return
}

void main( void)
{
  USHORT usOffsetDriver
       , usBytesLeft;                            // in search for NUL device
  PCH pchGlobal ;                                // pointer to system global data
  static CHAR szDriverName[] = "DEVHLPXX"        // device helper driver
        , szNullDriver[] = "NUL     ";           // first driver in system

  typedef struct _DDHEADER  {                    // device driver header
    struct _DDHEADER * pddNext ;                 // chain to next driver
    USHORT fsAttribute ;                         // driver attributes
    USHORT usStrategyEntryOffset ;
    USHORT usIDCEntryOffset ;                    // inter device communication
    CHAR chName[ 8] ;                            // name for character devices
    USHORT usIDCEntrySegmentProt ;
    USHORT usIDCDataSegmentProt ;
    USHORT usIDCEntrySegmentReal ;
    USHORT usIDCDataSegmentReal ;
  }  DDHEADER ;
  typedef DDHEADER * PDDHEADER ;
  PDDHEADER pddCurrent                           // current DCB
      , pddNext;                                 // next DCB
  SEL selDriver ;                                // selector of DCB

  // open the DEVHLP device
  if ((devhlp = open(szDriverName, 0)) == -1) {
    puts( "Can't find DEVHLP.SYS") ;
    exit( 1) ;
  }

  // locate the first driver
  selDriver = 0x50 ;                             // global data segment
  usOffsetDriver = 0 ;
  usBytesLeft = 32000 ;                          // should be large enough
  pchGlobal = MAKEP( MakeSel( selDriver), usOffsetDriver) ;
  do  {
    PCH pchMatch ;

    pchMatch = memchr( pchGlobal + 1, 'N', usBytesLeft); //look for first char
    if ( pchMatch == NULL)  {                    // if no match
      ReleaseSel( SELECTOROF( pchGlobal)) ;      // release the selector
      puts( "NUL driver not found") ;            // and give up
      exit( 1) ;
    }  // if no match
                        // partial match
    usBytesLeft -= pchMatch - pchGlobal ;        // reduce residual count
    pchGlobal = pchMatch ;                       // point to start of match
  }  while ( memcmp( pchGlobal                   // break out if name matches
          , szNullDriver                         //  exactly
          , sizeof szNullDriver - 1) != 0);

  // run the chain
  printf( "  Address Name\n") ;                        // column headings
  for ( usOffsetDriver = OFFSETOF( pchGlobal) - 0x0a // back up to DCB start
    , pddCurrent = ( PDDHEADER) ( pchGlobal - 0x0a)
    , selDriver = SELECTOROF( pddCurrent->pddNext)   // selector of next DCB
      ; ; ) {
    printf( "%4X:%04X ", selDriver, usOffsetDriver);
    if ( ( pddCurrent->fsAttribute & 0x8000) == 0)  // if block driver
      printf( "Block device, %d logical units\n"
            , pddCurrent->chName[ 0]);               // number of units
    else                                             // if character driver
      printf( "%-8.8s\n", pddCurrent->chName);
    selDriver = SELECTOROF( pddCurrent->pddNext) ;  // point to next DCB
    usOffsetDriver = OFFSETOF( pddCurrent->pddNext) ;
    if ( usOffsetDriver == 0xffff)                   // if end of chain
      break ;                                        // we are done
    pddNext = MAKEP( MakeSel( selDriver), usOffsetDriver) ;
    ReleaseSel( SELECTOROF( pddCurrent)) ;           // free previous DCB
    pddCurrent = pddNext ;                           // age the pointer
  }  // loop once for each device driver

  // release the last selector
  ReleaseSel( SELECTOROF( pddCurrent)) ;

  exit( 0) ;
}
```

End Listing Two

Listing Three

```
/* DEVHLP.H -- for use with DosDevIOCtl and DEVHLP.SYS */

#define DevHlp_SchedClockAddr       0x00
#define DevHlp_DevDone              0x01
#define DevHlp_Yield                0x02
#define DevHlp_TCYield              0x03
#define DevHlp_Block                0x04
#define DevHlp_Run                  0x05
#define DevHlp_SemRequest           0x06
#define DevHlp_SemClear             0x07
#define DevHlp_SemHandle            0x08
#define DevHlp_PushReqPacket        0x09
#define DevHlp_PullReqPacket        0x0A
#define DevHlp_PullParticular       0x0B
#define DevHlp_SortReqPacket        0x0C
#define DevHlp_AllocReqPacket       0x0D
#define DevHlp_FreeReqPacket        0x0E
#define DevHlp_QueueInit            0x0F
#define DevHlp_QueueFlush           0x10
#define DevHlp_QueueWrite           0x11
#define DevHlp_QueueRead            0x12
#define DevHlp_Lock                 0x13
#define DevHlp_Unlock               0x14
#define DevHlp_PhysToVirt           0x15
#define DevHlp_VirtToPhys           0x16
#define DevHlp_PhysToUVirt          0x17
#define DevHlp_AllocPhys            0x18
#define DevHlp_FreePhys             0x19
#define DevHlp_SetROMVector         0x1A
#define DevHlp_SetIRQ               0x1B
#define DevHlp_UnSetIRQ             0x1C
#define DevHlp_SetTimer             0x1D
#define DevHlp_ResetTimer           0x1E
#define DevHlp_MonitorCreate        0x1F
#define DevHlp_Register             0x20
#define DevHlp_DeRegister           0x21
#define DevHlp_MonWrite             0x22
#define DevHlp_MonFlush             0x23
#define DevHlp_GetDosVar            0x24
#define DevHlp_SendEvent            0x25
#define DevHlp_ROMCritSection       0x26
#define DevHlp_VerifyAccess         0x27
#define DevHlp_SysTrace             0x28
#define DevHlp_AttachDD             0x2A
#define DevHlp_AllocGDTSelector     0x2D
#define DevHlp_PhysToGDTSelector    0x2E
#define DevHlp_RealToProt           0x2F
#define DevHlp_ProtToReal           0x30
#define DevHlp_EOI                  0x31
#define DevHlp_UnPhysToVirt         0x32
#define DevHlp_TickCount            0x33
#define DevHlp_GetLIDEntry          0x34
#define DevHlp_FreeLIDEntry         0x35
#define DevHlp_ABIOSCall            0x36
#define DevHlp_ABIOSCommonEntry     0x37
#define DevHlp_RegisterStackUsage   0x38

#define UVirt_Exec                  0
#define UVirt_ReadWrite             1
#define UVirt_Release               2

#pragma pack(1)

typedef struct {
    unsigned int carry              : 1;
    unsigned int                    : 1;
    unsigned int parity             : 1;
    unsigned int                    : 1;
    unsigned int aux                : 1;
    unsigned int                    : 1;
    unsigned int zero               : 1;
    unsigned int sign               : 1;
    unsigned int trap               : 1;
    unsigned int int_en             : 1;
    unsigned int direction          : 1;
    unsigned int overflow           : 1;
    unsigned int iopl               : 2;
    unsigned int nest_task          : 1;
    unsigned int                    : 1;
    } FLAGS;

typedef struct {
    USHORT ax,bx,cx,dx,si,di,ds,es;
    FLAGS  flags;
    } REGS;
```

End Listings

Listing One *(Text begins on page 42.)*

```
            TITLE - BACKDOOR.SYS - closes DOS's backdoors
            PAGE 60,132
            .RADIX 16

; BACKDOOR.SYS closes two "backdoors" into the MS-DOS INT 21h function
; dispatcher that could be used by a virus or trojan horse to cause damage.
; It also filters INT 21h directly to reject a special case of function 13h
; which could destroy all data on a disk.
; For use with MASM 5.1
; MASM BACKDOOR;
; LINK BACKDOOR;
; EXE2BIN BACKDOOR.EXE BACKDOOR.SYS
;
       ASSUME CS:CSEG, DS:CSEG
CSEG   SEGMENT PARA PUBLIC 'CODE'
            ORG 0000h                   ; device driver starts at 0

            DW 0FFFFh,0FFFFh            ; far pointer to next device
            DW 8000h                    ; character device driver
            DW offset DEV_STRAT_RTN     ; pointer to the strategy routine
            DW offset DEV_INT_RTN       ; pointer to the interrupt routine
            DB "B"+80h,"ACKDOOR"        ; device name with high bit set will
                                        ; avoid any filename conflicts
INSTALL_MSG DB 0Dh,0Ah
            DB "BACKDOOR is installed at $"

DEV_HDR_BX  DW 0000                     ; pointer for ES:BX for device
DEV_HDR_ES  DW 0000                     ; request header

ORIG_INT21_OFF DW 0000                  ;
ORIG_INT21_SEG DW 0000                  ;

TEMP        DW 0000                     ; used for temporary storage

REFUSE_RQST PROC FAR                    ;
            POP AX                      ; get rid of flags on stack
            POP AX                      ; get the return segment
            POP CS:TEMP                 ; and save offset
            PUSH AX                     ; save the return address in proper
            PUSH CS:TEMP                ; order
            STC                         ; return STC for error
            MOV AX,0FFFFh               ; return AX=-1
            RET                         ; and do FAR RET back to caller
REFUSE_RQST ENDP

NEW_INT21   PROC NEAR
            PUSH AX                     ; save original registers first thing
            PUSH BX
            CMP AH,13h                  ; is this the DELETE FCB function?
            JNZ CONT_ORIG_INT21         ; no, so continue on
            MOV BX,DX                   ; point BX to the FCB
            CMP byte ptr DS:[BX],0FFh ; got an extended FCB?
            JNZ CONT_ORIG_INT21         ; no, so continue on
            CMP byte ptr DS:[BX+6],1Fh; yes, so got the special attribute?
            JNZ CONT_ORIG_INT21         ; no, so continue on
            CMP word ptr DS:[BX+8],"??"; yes, so filename starts with "??" ?
            JNZ CONT_ORIG_INT21         ; no, so continue on
            CMP word ptr DS:[BX+0Ah],"??"; yes, so filename = "??" ?
            JNZ CONT_ORIG_INT21         ; no, so continue on
            CMP word ptr DS:[BX+0Ch],"??"; yes, so filename = "??" ?
            JNZ CONT_ORIG_INT21         ; no, so continue on
            CMP word ptr DS:[BX+0Eh],"??"; yes, so filename = "??" ?
            JNZ CONT_ORIG_INT21         ; no, so continue on
            CMP word ptr DS:[BX+10h],"??"; yes, so filename = "??" ?
            JNZ CONT_ORIG_INT21         ; no, so continue on
            CMP byte ptr DS:[BX+12h],"?"; yes, so filename ends with "??" ?
            JNZ CONT_ORIG_INT21         ; no, so continue on
            POP BX                      ; yes, so reject it altogether
            POP AX                      ;
            MOV AL,0FFh                 ; return match not found
            STC                         ; STC just for the heck of it
            RETF 0002                   ; and IRET with new flags

CONT_ORIG_INT21:
            POP BX                      ; restore original registers
            POP AX                      ;
            JMP dword ptr CS:ORIG_INT21_OFF; continue with original handler
NEW_INT21   ENDP

DEV_STRAT_RTN PROC FAR                  ;
            MOV CS:DEV_HDR_BX,BX        ; save the ES:BX pointer to the
            MOV CS:DEV_HDR_ES,ES        ; device request header
            RET                         ;
DEV_STRAT_RTN ENDP

DEV_INT_RTN  PROC FAR                   ;
            PUSH AX                     ; save all registers
            PUSH BX                     ;
            PUSH CX                     ;
            PUSH DX                     ;
            PUSH DS                     ;
            PUSH ES                     ;
            PUSH DI                     ;
            PUSH SI                     ;
            PUSH BP                     ;
            PUSH CS                     ;
            POP DS                      ; point DS to local code
            LES DI,dword ptr DEV_HDR_BX; ES:DI=device request header
            MOV BL,ES:[DI+02]           ; get the command code
            XOR BH,BH                   ; clear out high byte
            CMP BX,00h                  ; doing an INSTALL?
            JNZ DEV_IGNORE              ; no, so just ignore the call then
            CALL INSTALL_BACKDOOR       ; yes, so install code in memory

DEV_IGNORE:                             ;
            MOV AX,0100h                ; return STATUS of DONE
            LDS BX,dword ptr CS:DEV_HDR_BX; DS:BX=device request header
            MOV [BX+03],AX              ; return STATUS in the header
            POP BP                      ; restore original registers
            POP SI                      ;
            POP DI                      ;
            POP ES                      ;
            POP DS                      ;
            POP DX                      ;
            POP CX                      ;
            POP BX                      ;
            POP AX                      ;
            RET                         ; and RETF to DOS
DEV_INT_RTN  ENDP

INSTALL_BACKDOOR PROC NEAR              ;
            CALL CLOSE_BACK_DOOR        ; install new handler to close back
                                        ; door
            CALL HOOK_INT21             ; and hook INT21 filter
            MOV AH,09h                  ; DOS display string
            MOV DX,offset INSTALL_MSG ; show installation message
            INT 21h                     ; via DOS
            MOV AX,CS                   ; display current code segment
            CALL OUTPUT_AX_AS_HEX       ; output AX as two HEX digits
            MOV AL,3Ah                  ; now output a colon
            CALL DISPLAY_TTY            ; to the screen
            MOV AX,offset REFUSE_RQST ; show new handler's offset
            CALL OUTPUT_AX_AS_HEX       ; output AX as two HEX digits
            CALL DISPLAY_NEWLINE        ; output a newline to finish display
            LES DI,dword ptr DEV_HDR_BX; ES:DI=device request header
            MOV Word Ptr ES:[DI+0Eh],offset INSTALL_BACKDOOR; this is the
            MOV ES:[DI+10h],CS          ; end of resident code
            RET                         ;
INSTALL_BACKDOOR ENDP

CLOSE_BACK_DOOR PROC NEAR               ;
            PUSH ES                     ; save original registers
            PUSH AX                     ;
            PUSH BX                     ;
            XOR AX,AX                   ; point ES to the interrupt vector
            MOV ES,AX                   ; table
            MOV BX,00C1h                ; install new handler at INT30 + 1
            MOV AX,offset REFUSE_RQST ; get new offset for the handler
            MOV ES:[BX],AX              ; save it in interrupt vector table
            MOV AX,CS                   ; get the segment for the handler
            MOV ES:[BX+02],AX           ; and save it, too
            POP BX                      ; restore original registers
            POP AX                      ;
            POP ES                      ;
            RET                         ; and RET to caller
CLOSE_BACK_DOOR ENDP

HOOK_INT21   PROC NEAR
            PUSH AX
            PUSH BX
            PUSH ES
            MOV AX,3521h                ; get current INT21 vector
            INT 21h                     ; via DOS
```

```
                MOV CS:ORIG_INT21_OFF,BX  ; save the offset
                MOV BX,ES                 ;
                MOV CS:ORIG_INT21_SEG,BX  ; and the segment
                PUSH CS
                POP DS                    ; make sure DS=local code
                MOV DX,offset NEW_INT21   ; point to new handler
                MOV AX,2521h              ; install new handler
                INT 21h                   ; via DOS
                POP ES                    ; and restore original registers
                POP BX
                POP AX
                RET                       ; and RET to caller
HOOK_INT21      ENDP

OUTPUT_AX_AS_HEX PROC NEAR                ;
                PUSH AX                   ; save original registers
                PUSH BX                   ;
                PUSH CX                   ;
                PUSH AX                   ; save number for output
                MOV AL,AH                 ; output high byte first
                CALL OUTPUT_AL_AS_HEX     ; output AL as two HEX digits
                POP AX                    ; output low byte next
                CALL OUTPUT_AL_AS_HEX     ; output AL as two HEX digits
                POP CX                    ; restore original registers
                POP BX                    ;
                POP AX                    ;
                RET                       ; and RET to caller
OUTPUT_AX_AS_HEX ENDP

OUTPUT_AL_AS_HEX PROC NEAR                ;
                PUSH AX                   ; save original registers
                PUSH BX                   ;
                PUSH CX                   ;

                PUSH AX                   ; save the number for output (in AL)
                MOV CL,04h                ; first output high nibble
                SHR AL,CL                 ; get digit into low nibble
                ADD AL,30h                ; convert to ASCII
                CMP AL,39h                ; got a decimal digit?
                JBE OUTPUT_FIRST_DIGIT    ; yes, so continue
                ADD AL,07h                ; no, so convert to HEX ASCII

OUTPUT_FIRST_DIGIT:                       ;
                CALL DISPLAY_TTY          ; output it via BIOS
                POP AX                    ; get number back
                AND AL,0Fh                ; keep only low digit now
                ADD AL,30h                ; convert to ASCII
                CMP AL,39h                ; got a decimal digit?
                JBE OUTPUT_SECOND_DIGIT   ; yes, so continue
                ADD AL,07h                ; no, so convert to HEX ASCII

OUTPUT_SECOND_DIGIT:
                CALL DISPLAY_TTY          ; output it via BIOS
                POP CX                    ; restore original registers
                POP BX                    ;
                POP AX                    ;
                RET                       ; and RET to caller
OUTPUT_AL_AS_HEX ENDP

DISPLAY_NEWLINE PROC NEAR                 ;
                PUSH AX                   ; save original AX
                MOV AL,0Dh                ; first do CR
                CALL DISPLAY_TTY          ; output it via the BIOS
                MOV AL,0Ah                ; do LF next
                CALL DISPLAY_TTY          ; output it via the BIOS
                POP AX                    ; restore original AX
                RET                       ; and RET to caller
DISPLAY_NEWLINE ENDP

DISPLAY_TTY     PROC NEAR                 ;
                PUSH AX                   ;
                PUSH BX                   ;
                MOV AH,0Eh                ; display TTY
                MOV BX,0007h              ; on page 0, normal attribute
                INT 10h                   ; via BIOS
                POP BX                    ;
                POP AX                    ;
                RET                       ;
DISPLAY_TTY     ENDP

CSEG    ENDS
        END
```

End Listings

Listing One *(Text begins on page 50.)*

```
/* The following is a RAM disk driver developed for Unix Sys V/386
 * release 3.2. -- Author: Jeff Reagen          05-02-90.
*/
#include "sys/types.h"
#include "sys/param.h"
#include "sys/immu.h"
#include "sys/fs/s5dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/cmn_err.h"
#include "sys/buf.h"

#define RD_SIZE_IN_PAGES 0x100L          /* 256 4K pages => 1 MB     */
#define RD_MAX           1               /* Max RAM Disks            */
#define RAMDISK(x)       (int)(x&0x0F)   /* Ram disk number from dev */
#define DONT_SLEEP       1               /* sptalloc parameter       */

/* For ioctl routines.
*/
#define RD_GETSIZE      1                /* return size of RAM disk     */
struct rd_getsize {                      /* Structure passed to rdioctl */
   daddr_t sectors;
   long    in_bytes;
};

/* Valid states for the RAM disk driver.
*/
#define RD_UNDEFINED     0x0000          /* Disk has not been setup */
#define RD_CONFIGURED    0x0001          /* Configured disk */
#define RD_OPEN          0x0002          /* Indicates disk has been opened */

/*   The RAM disk is created iff the size field has been defined. Since
 *   sptalloc only allocates pages, make sure the size is
 *   some multiple of page size (4096).
*/
struct ram_config {
   int      state;     /* current state               */
   caddr_t  virt;      /* virtual address of RAM disk  */
   long     size;      /* RAM disk size in units of 4K */
};

struct ram_config rd_cfg = {RD_UNDEFINED, (caddr_t)0, RD_SIZE_IN_PAGES};

extern caddr_t sptalloc();

/* rdinit - initialize the RAM disk.
 */
rdinit (dev)
   dev_t   dev;
{
   /* Has a RAM disk been defined? */
   if (rd_cfg.size == 0)
   {
      /* Just return silently - ram disk is not configured. */
      return 0;
   }

   /* Last parameter 1 in sptalloc calls prevents sleep if no memory. */
   if ((rd_cfg.virt = sptalloc (rd_cfg.size, PG_P,0,DONT_SLEEP)) == NULL)
   {
      cmn_err (CE_WARN,"Could not allocate enough memory for RAM disk.\n");
      return 0;
   }
   rd_cfg.state |= RD_CONFIGURED;

   return;
}

/*  rdopen
 */
rdopen (dev)
   dev_t dev;
{
   int rdisk;

   rdisk = RAMDISK(dev);

   if ( rdisk >= RD_MAX)
   {
      /* RAM disk specified foes not exist. */
      u.u_error = ENODEV;
      return;
   }

   /* Make sure ram disk has been configured. */
   if ( (rd_cfg.state & RD_CONFIGURED) != RD_CONFIGURED)
   {
      /* disk has not been configured! */
      u.u_error = ENOMEM;
      return;
   }

   /* RAM disk successfully opened. */
   rd_cfg.state |= RD_OPEN;
}
/*  rdclose - close the RAM disk.
 */
rdclose (dev)
   dev_t   dev;
{
   rd_cfg.state &= ~RD_OPEN;
   return;
}

/*  rdstrategy - the entire synchronous transfer operation happens here.
 */
rdstrategy (bp)
   register struct buf *bp;
{
   register long  req_start;      /* start of transfer */
   register long  byte_size;      /* Max capacity of RAM disk in bytes. */
            int   disk;           /* RAM disk being requested for service. */

   disk = RAMDISK(bp->b_dev);

   /* Validate disk number. */
   if (disk >= RD_MAX)
   {
         /* Disk does not exist. */
         bp->b_flags |= B_ERROR;
         bp->b_error = ENODEV;
         iodone(bp);
         return;
   }

   /* Validate request range. Reads can be trimmed back... */
   byte_size = rd_cfg.size * NBPP;
   req_start = bp->b_blkno * NBPSCTR;
   bp->b_resid = 0;           /* Number of bytes remaining after transfer */

   /* Check for requests exceeding the upper bound of the disk. */
   if (req_start + bp->b_bcount > byte_size)
   {
      if (bp->b_flags & B_READ)
      {
         /* Read */
         /* Adjust residual count. */
         bp->b_resid = req_start +  bp->b_bcount - byte_size;
         bp->b_bcount = byte_size - req_start;
      }
      else
      {
         /* Write - always fails */
         bp->b_resid = bp->b_bcount;
         bp->b_flags |= B_ERROR;
         iodone (bp);
         return;
      }
   }

   /* Service the request. */
   if (bp->b_flags & B_READ)
   {
      bcopy (rd_cfg.virt + req_start, bp->b_un.b_addr, bp->b_bcount);
   }
   else
   {
      bcopy (bp->b_un.b_addr, rd_cfg.virt + req_start, bp->b_bcount);
   }
   bp->b_flags &= ~B_ERROR;    /* Make sure an error is NOT reported. */
   iodone(bp);
   return;
}
```

```
/*  rdread - character read interface.
*/
rdread (dev)
   dev_t   dev;
{
   /* Validate request based on number of 512 bytes sectors supported. */
   if (physck ((daddr_t)rd_cfg.size << DPPSHFT, B_READ))
   {
      /* Have physio allocate the buffer header, then call rdstrategy. */
      physio (rdstrategy, (struct buf *)NULL, dev, B_READ);
   }
}

/*  rdwrite - character write interface.
*/
rdwrite (dev)
   dev_t   dev;
{
   /* Validate request based on number of 512 bytes sectors supported. */
   if (physck ((daddr_t)rd_cfg.size << DPPSHFT, B_WRITE))
   {
      /* Have physio allocate the buffer header, then call rdstrategy. */
      physio (rdstrategy, (struct buf *)NULL, dev, B_WRITE);
   }
}

/*   rdioctl - returns size of RAM disk.
 */
rdioctl (dev, command, arg, mode)
   dev_t   dev;
   int     command;
   int     *arg;
   int     mode;
{
   struct  rd_getsize sizes;

   if ( RAMDISK(dev) > RD_MAX || !(rd_cfg.state&RD_CONFIGURED) )
   {
      u.u_error = ENODEV;
      return;
   }

   switch (command) {
      case RD_GETSIZE:
         sizes.sectors = rd_cfg.size << DPPSHFT;
         sizes.in_bytes = rd_cfg.size * NBPP;
         /* Now transfer the request to user space */
         if (copyout (&sizes, arg, sizeof (sizes)) )
         {
            u.u_error = EFAULT;
         }
         break;

      default:
         /* Error - do not recognize command submitted. */
         u.u_error = EINVAL;
         return;
   }
}

/*   rdintr - the RAM disk does not generate hardware interrupts,
 *   so this routine simply prints a warning message and returns.
 */
rdintr ()
{
   cmn_err (CE_WARN, "RAM disk took a spurious hardware interrupt.\n");
}

/*  rdprint - send messages concerning the RAM disk to the console.
 */
rdprint (dev, str)
   dev_t  dev;
   char   *str;
{
   cmn_err (CE_NOTE, "%s on Ram Disk %d.\n", str, RAMDISK (dev));
}
```

End Listing

Listing One *(Text begins on page 58.)*

```
/* min.c -- Procedure to find the smallest element of the array.
   The number of elements in the array must be > 0.  ( n > 0 ) */
float find_min( array, n )
float array[];          /* input array */
int n;                  /* number of elements in the array ( n > 0 ) */
{
  register i;  float min;

  min = array[0];
  for( i = 1; i < n; i++ )
     if ( min > array[i] )   min = array[i];
  return( min );
}
/* Procedure to find the largest element of the array.
   The number of elements in the array must be > 0.  ( n > 0 ) */
float find_max( array, n )
float array[];          /* input array */
int n;                  /* number of elements in the array ( n > 0 ) */
{
  register i;  float max;

  max = array[0];
  for( i = 1; i < n; i++ )
     if ( max < array[i] )   max = array[i];
  return( max );
}
```

End Listing One

Listing Two

```
/* min_max.c  -- Procedure to find the smallest and the largest element of
   the array. The number of elements in the array must be > 0.  ( n > 0 ) */
void find_min_max( array, n, min, max )
float array[];          /* input array */
int n;                  /* number of elements in the array ( n > 0 ) */
float *min, *max;       /* pointers to the return values */
{
  register i;

  if ( n <= 1 )
     *min = *max = array[0];
  else {
     if ( array[0] > array[1] ) {    /* establish the basis min and max */
        *max = array[0];
        *min = array[1];
     }
     else {
        *max = array[1];
        *min = array[0];
     }
     for( i = 2; ( i + 2 ) <= n; i += 2 )
        if ( array[ i ] > array[ i + 1 ] ) {
           if ( array[ i     ] > *max )   *max = array[ i ];
           if ( array[ i + 1 ] < *min )   *min = array[ i + 1 ];
        }
        else {
           if ( array[ i + 1 ] > *max )   *max = array[ i + 1 ];
           if ( array[ i     ] < *min )   *min = array[ i ];
        }
     if ( i < n )        /* handle the odd/last array element */
       if ( *max < array[i] )  *max = array[i];
       else if ( *min > array[i] )  *min = array[i];
  }
}
```

End Listings

Listing One *(Text begins on page 62.)*

```
#include <stdio.h>
long area[3];
void compute_areas(void);

main()
{
        int i;
        extern long area[3];
        compute_areas();
        for(i = 0; i <= 2; i++) printf("The approximate area of function %d is
                                                        %ld\n", i, area[i]);
}

void compute_areas(void)
{       long temp, y0[100], y1[100], y2[100];
        extern long area[3];
        int i, k, a, b, c, d, n, m, j;

        a = 2; b = 5; c = 1; d = 3;
        area[0] = area[1] = area[2] = temp = 0;

       /*  compute 100 initial y values for all functions and approximate area
        under curves. Do this 100 times. */
        for (i = 0; i <= 99; i++) {
                for (j = 0; j <= 99; j++) {
                        n = j-1;
                        m = j+1;
                        y0[j] = a*j*n + 48*j - (c+d);
                        y1[j] = d*(j+10) + a*b;
                        y2[j] = c*j*m + 73*j*j + n;
                }
        }
/* add a*b+c+d to all y values. Do this 100 times. */
        for (i = 0; i <= 99; i++) {
                for (j = 0; j <= 99; j++) {
                        y0[j] += a*b+c+d;
                        y1[j] += a*b+c+d;
                        y2[j] += a*b+c+d;
                }
        }

/* bubblesort each array */
        for (i = 0; i <= 99; i++) {
                for (k = 99; k >= 1; k--) {
                        if (y0[k-1] > y0[k]) {
                                temp = y0[k];
                                y0[k] = y0[k-1];
                                y0[k-1] = temp;
                        }
                        if (y1[k-1] > y1[k]) {
                                temp = y1[k];
                                y1[k] = y1[k-1];
                                y1[k-1] = temp;
                        }
                        if (y2[k-1] > y2[k]) {
                                temp = y2[k];
                                y2[k] = y2[k-1];
                                y2[k-1] = temp;
                        }
                }
        }

        /*  now compute areas */
        for (j = 0; j <= 99; j++) {
                area[0] += y0[j];
                area[1] += y1[j];
                area[2] += y2[j];
        }

        return;
   }
```

End Listing One

Listing Two

```
#include <stdio.h>
long area[3];
void compute_areas(void);

main()
{
        int i;
        extern long area[3];
        compute_areas();
        for(i = 0; i <= 2; i++) printf("The approximate area of function %d is
                                                        %ld\n", i, area[i]);

}
void compute_areas(void)
{       long temp, y0[100], y1[100], y2[100];
        extern long area[3];
        int i, k, a, b, c, d, n, m, j;

        a = 2; b = 5; c = 1; d = 3;
        area[0] = area[1] = area[2] = temp = 0;

 /*  compute 100 initial y values for all functions and approximate area under
        curves. Do this 100 times. */
        for (i = 0; i <= 99; i++) {
                for (j = 0; j <= 99; j++) {
                        n = j-1;
                        m = j+1;
                        y0[j] = a*j*n + 48*j - 4;
                        y1[j] = d*j +40;
                        y2[j] = c*j*j*m - 73*j*j;
                }
        }

/* add 14 to all y values. Do this 100 times. */
        for (i = 0; i <= 99; i++) {
                for (j = 0; j <= 99; j++) {
                        y0[j] += 14;
                        y1[j] += 14;
                        y2[j] += 14;
                }
        }

/* bubblesort each array */
        for (i = 0; i <= 99; i++) {
                for (k = 99; k >= 1; k--) {
                        if (y0[k-1] > y0[k]) {
                                temp = y0[k];
                                y0[k] = y0[k-1];
                                y0[k-1] = temp;
                        }
                        if (y1[k-1] > y1[k]) {
                                temp = y1[k];
                                y1[k] = y1[k-1];
                                y1[k-1] = temp;
                        }
                        if (y2[k-1] > y2[k]) {
                                temp = y2[k];
                                y2[k] = y2[k-1];
                                y2[k-1] = temp;
                        }
                }
        }

        /*  now compute areas */
        for (j = 0; j <= 99; j++) {
                area[0] += y0[j];
                area[1] += y1[j];
                area[2] += y2[j];
        }

        return;
}
```

End Listing Two

Listing Three

```
#include <stdio.h>
long area[3];
void compute_areas(void);

main()
{
        int i;
        extern long area[3];
        compute_areas();
        for(i = 0; i <= 2; i++) printf("The approximate area of function %d is
                                                        %ld\n", i, area[i]);
}

void compute_areas(void)
{       long temp, y0[100], y1[100], y2[100];
        extern long area[3];
        int i, k, j;

        area[0] = area[1] = area[2] = temp = 0;

 /*  compute 100 initial y values for all functions and approximate area under
        curves. Do this 100 times. */
        for (i = 0; i <= 99; i++) {
                for (j = 0; j <= 99; j++) {
                        y0[j] = 2*j*j - 50*j +10;
                        y1[j] = 3*j + 54;
                        y2[j] = j*j*j - 72*j + 14;
                }
        }

/* bubblesort each array */
        for (i = 0; i <= 99; i++) {
                for (k = 99; k >= 1; k--) {
                        if (y0[k-1] > y0[k]) {
                                temp = y0[k];
                                y0[k] = y0[k-1];
                                y0[k-1] = temp;
                        }
                        if (y1[k-1] > y1[k]) {
                                temp = y1[k];
                                y1[k] = y1[k-1];
                                y1[k-1] = temp;
                        }
                        if (y2[k-1] > y2[k]) {
                                temp = y2[k];
                                y2[k] = y2[k-1];
                                y2[k-1] = temp;
                        }
                }
        }

        /*  now compute areas */
        for (j = 0; j <= 99; j++) {
                area[0] += y0[j];
                area[1] += y1[j];
                area[2] += y2[j];
        }

        return;
}
```

End Listings

Listing Ten *(Text begins on page 72.)*

```
IMPLEMENTATION MODULE Term; (* TVI950 Terminal Emulation for Kermit *)

    FROM Drives IMPORT
       SetDrive;

    FROM Directories IMPORT
      FileAttributes, AttributeSet, DirectoryEntry, FindFirst, FindNext;

    FROM SYSTEM IMPORT
       ADR;

    FROM OS2DEF IMPORT
       ULONG;

    FROM DosCalls IMPORT
       DosChDir, DosSleep;

    FROM Screen IMPORT
       ClrScr, ClrEol, GotoXY, GetXY,
       Right, Left, Up, Down, WriteAtt, WriteString, WriteLn, Write,
       attribute, NORMAL, HIGHLIGHT, REVERSE;

    FROM PMWIN IMPORT
       WinPostMsg, MPFROM2SHORT;

    FROM Shell IMPORT
       comport, FrameWindow;

    FROM KH IMPORT
       COM_OFF;

    FROM CommPort IMPORT
       CommStatus, GetChar, SendChar;

    FROM Strings IMPORT
       Length, Concat;

    IMPORT ASCII;


    CONST
    (* Key codes:  Note: F1 -- F12 are actually Shift-F1 -- Shift-F12 *)
       F1 = 124C;
       F2 = 125C;
       F3 = 126C;
       F4 = 127C;
       F5 = 130C;
       F6 = 131C;
       F7 = 132C;
       F8 = 133C;
       F9 = 134C;
       F10 = 135C;
       F11 = 207C;
       F12 = 210C;
       AF1 = 213C;    (* Alt-F1 *)
       AF2 = 214C;    (* Alt-F2 *)
       INS = 122C;
       DEL = 123C;
       HOME = 107C;
       PGDN = 121C;    (* synonym for PF10 *)
       PGUP = 111C;    (* synonym for PF11 *)
       ENDD = 117C;    (* synonym for PF12 *)
       UPARROW = 110C;
       DOWNARROW = 120C;
       LEFTARROW = 113C;
       RIGHTARROW = 115C;
       CtrlX = 30C;
       CtrlCaret = 36C;
       CtrlZ = 32C;
       CtrlL = 14C;
       CtrlH = 10C;
       CtrlK = 13C;
       CtrlJ = 12C;
       CtrlV = 26C;
       ESC = 33C;
       BUFSIZE = 4096;    (* character buffer used by term thread *)


    VAR
       commStat : CommStatus;
       echo : (Off, Local, On);
       newline: BOOLEAN;    (* translate <cr> to <cr><lf> *)
       Insert : BOOLEAN;


    PROCEDURE Dir (path : ARRAY OF CHAR);
    (* Change drive &/or directory; display directory (in wide format) *)

       VAR
          gotFN : BOOLEAN;
          filename : ARRAY [0..20] OF CHAR;
          attr : AttributeSet;
          ent : DirectoryEntry;
          i, j, k : INTEGER;

       BEGIN
          filename := "";    (* in case no directory change *)
          i := Length (path);
          IF (i > 2) AND (path[1] = ':') THEN    (* drive specifier *)
             DEC (i, 2);
             SetDrive (ORD (CAP (path[0])) - ORD ('A'));
             FOR j := 0 TO i DO    (* strip off the drive specifier *)
                path[j] := path[j + 2];
             END;
          END;
          IF i # 0 THEN
             gotFN := FALSE;
             WHILE (i >= 0) AND (path[i] # '\') DO
                IF path[i] = '.' THEN
                   gotFN := TRUE;
                END;
                DEC (i);
             END;
             IF gotFN THEN
                j := i + 1;
                k := 0;
                WHILE path[j] # 0C DO
                   filename[k] := path[j];
                   INC (k);       INC (j);
                END;
                filename[k] := 0C;
                IF (i = -1) OR ((i = 0) AND (path[0] = '\')) THEN
                   INC (i);
                END;
                path[i] := 0C;
             END;
          END;
          IF Length (path) # 0 THEN
             DosChDir (ADR (path), 0);
          END;
          IF Length (filename) = 0 THEN
             filename := "*.*";
          END;
          attr := AttributeSet {ReadOnly, Directory, Archive};
          i := 1;    (* keep track of position on line *)

          ClrScr;
          gotFN := FindFirst (filename, attr, ent);
          WHILE gotFN DO
             WriteString (ent.name);
             j := Length (ent.name);
             WHILE j < 12 DO (* 12 is max. length for "filename.typ" *)
                Write (' ');
                INC (j);
             END;
             INC (i);    (* next position on this line *)
             IF i > 5 THEN
                i := 1;    (* start again on new line *)
                WriteLn;
             ELSE
                WriteString (" : ");
             END;
             gotFN := FindNext (ent);
          END;
          WriteLn;
       END Dir;


    PROCEDURE InitTerm;
    (* Clear Screen, Home Cursor, Get Ready For Terminal Emulation *)
       BEGIN
          ClrScr;
          Insert := FALSE;
          attribute := NORMAL;
       END InitTerm;


    PROCEDURE PutKbdChar (ch1, ch2 : CHAR);
    (* Process a character received from the keyboard *)
       BEGIN
          IF ch1 = ASCII.enq THEN    (* Control-E *)
             echo := On;
          ELSIF ch1 = ASCII.ff THEN    (* Control-L *)
             echo := Local;
          ELSIF ch1 = ASCII.dc4 THEN    (* Control-T *)
             echo := Off;
          ELSIF ch1 = ASCII.so THEN    (* Control-N *)
             newline := TRUE;
          ELSIF ch1 = ASCII.si THEN    (* Control-O *)
             newline := FALSE;
          ELSIF (ch1 = ASCII.can) OR (ch1 = ESC) THEN
             attribute := NORMAL;
             WinPostMsg (FrameWindow, WM_TERMQUIT, 0, 0);
          ELSIF ch1 = 0C THEN
             Function (ch2);
          ELSE
             commStat := SendChar (comport - COM_OFF, ch1, FALSE);
             IF (echo = On) OR (echo = Local) THEN
                WriteAtt (ch1);
             END;
          END;
       END PutKbdChar;


    PROCEDURE Function (ch : CHAR);
    (* handles the function keys -- including PF1 - PF12, etc. *)
     BEGIN
       CASE ch OF
         F1 :  commStat := SendChar (comport - COM_OFF, ASCII.soh, FALSE);
               commStat := SendChar (comport - COM_OFF, '@', FALSE);
               commStat := SendChar (comport - COM_OFF, ASCII.cr, FALSE);
       | F2  :  commStat := SendChar (comport - COM_OFF, ASCII.soh, FALSE);
               commStat := SendChar (comport - COM_OFF, 'A', FALSE);
               commStat := SendChar (comport - COM_OFF, ASCII.cr, FALSE);
       | F3  :  commStat := SendChar (comport - COM_OFF, ASCII.soh, FALSE);
               commStat := SendChar (comport - COM_OFF, 'B', FALSE);
               commStat := SendChar (comport - COM_OFF, ASCII.cr, FALSE);
       | F4  :  commStat := SendChar (comport - COM_OFF, ASCII.soh, FALSE);
               commStat := SendChar (comport - COM_OFF, 'C', FALSE);
               commStat := SendChar (comport - COM_OFF, ASCII.cr, FALSE);
       | F5  :  commStat := SendChar (comport - COM_OFF, ASCII.soh, FALSE);
               commStat := SendChar (comport - COM_OFF, 'D', FALSE);
               commStat := SendChar (comport - COM_OFF, ASCII.cr, FALSE);
       | F6  :  commStat := SendChar (comport - COM_OFF, ASCII.soh, FALSE);
               commStat := SendChar (comport - COM_OFF, 'E', FALSE);
               commStat := SendChar (comport - COM_OFF, ASCII.cr, FALSE);
       | F7  :  commStat := SendChar (comport - COM_OFF, ASCII.soh, FALSE);
               commStat := SendChar (comport - COM_OFF, 'F', FALSE);
```

(continued on page 110)

Listing Ten *(Listing continued, text begins on page 72.)*

```
                commStat := SendChar (comport - COM_OFF, ASCII.cr, FALSE);
      |  F8  :  commStat := SendChar (comport - COM_OFF, ASCII.soh, FALSE);
                commStat := SendChar (comport - COM_OFF, 'G', FALSE);
                commStat := SendChar (comport - COM_OFF, ASCII.cr, FALSE);
      |  F9  :  commStat := SendChar (comport - COM_OFF, ASCII.soh, FALSE);
                commStat := SendChar (comport - COM_OFF, 'H', FALSE);
                commStat := SendChar (comport - COM_OFF, ASCII.cr, FALSE);
      |  F10,
         PGDN: commStat := SendChar (comport - COM_OFF, ASCII.soh, FALSE);
                commStat := SendChar (comport - COM_OFF, 'I', FALSE);
                commStat := SendChar (comport - COM_OFF, ASCII.cr, FALSE);
      |  F11,
         AF1,
         PGUP: commStat := SendChar (comport - COM_OFF, ASCII.soh, FALSE);
                commStat := SendChar (comport - COM_OFF, 'J', FALSE);
                commStat := SendChar (comport - COM_OFF, ASCII.cr, FALSE);
    |  F12,
       AF2,
       ENDD: commStat := SendChar (comport - COM_OFF, ESC, FALSE);
             commStat := SendChar (comport - COM_OFF, 'Q', FALSE);
    | INS : IF NOT Insert THEN
                commStat := SendChar (comport - COM_OFF, ESC, FALSE);
                commStat := SendChar (comport - COM_OFF, 'E', FALSE);
             END;
    | DEL : commStat := SendChar (comport - COM_OFF, ESC, FALSE);
             commStat := SendChar (comport - COM_OFF, 'R', FALSE);
    | HOME  : commStat := SendChar (comport - COM_OFF, CtrlZ, FALSE);
    | UPARROW  : commStat := SendChar (comport - COM_OFF, CtrlK, FALSE);
    | DOWNARROW : commStat := SendChar (comport - COM_OFF, CtrlV, FALSE);
    | LEFTARROW : commStat := SendChar (comport - COM_OFF, CtrlH, FALSE);
    | RIGHTARROW : commStat := SendChar (comport - COM_OFF, CtrlL, FALSE);
    ELSE
    (* do nothing *)
    END;
  END Function;


    PROCEDURE TermThrProc;
    (* Thread to get characters from port, put into buffer *)

       VAR
          ch : CHAR;

       BEGIN
          LOOP
             IF GetChar (comport - COM_OFF, ch) = Success THEN
       WinPostMsg (FrameWindow, WM_TERM, MPFROM2SHORT (ORD (ch), 0), 0);
             ELSE
                DosSleep (0);
             END
          END;
       END TermThrProc;


    VAR
       EscState, CurState1, CurState2 : BOOLEAN;
       CurChar1 : CHAR;

    PROCEDURE PutPortChar (ch : CHAR);
    (* Process a character received from the port *)
       BEGIN
          IF EscState THEN
             EscState := FALSE;
             IF ch = '=' THEN
                CurState1 := TRUE;
             ELSE
                Escape (ch);
             END;
          ELSIF CurState1 THEN
             CurState1 := FALSE;
             CurChar1 := ch;
             CurState2 := TRUE;
          ELSIF CurState2 THEN
             CurState2 := FALSE;
             Cursor (ch);
          ELSE
             CASE ch OF
                CtrlCaret, CtrlZ : ClrScr;
             |  CtrlL : Right;
             |  CtrlH : Left;
             |  CtrlK : Up;
             |  CtrlJ : Down;
             |  ESC   : EscState := TRUE;
             ELSE
                WriteAtt (ch);
                IF newline AND (ch = ASCII.cr) THEN
                   WriteLn;
                END;
             END;
          END;
          IF echo = On THEN
             commStat := SendChar (comport - COM_OFF, ch, FALSE);
          END;
       END PutPortChar;


    PROCEDURE Escape (ch : CHAR);
    (* handles escape sequences *)
       BEGIN
          CASE ch OF
             '*' : ClrScr;
          |  'T', 'R' : ClrEol;
          |  ')' : attribute := NORMAL;
          |  '(' : attribute := HIGHLIGHT;
          |  'f' : InsertMsg;
          |  'g' : InsertOn;
          ELSE
             (* ignore *)
          END;
       END Escape;


    PROCEDURE Cursor (ch : CHAR);
    (* handles cursor positioning *)

       VAR
          x, y : CARDINAL;

       BEGIN
          y := ORD (CurChar1) - 20H;
          x := ORD (ch) - 20H;
          GotoXY (x, y);   (* adjust for HOME = (1, 1) *)
       END Cursor;


    VAR
       cx, cy : CARDINAL;

    PROCEDURE InsertMsg;
    (* get ready insert mode; place message at bottom of the screen *)
       BEGIN
          IF NOT Insert THEN
             GetXY (cx, cy);   (* record current position *)
             GotoXY (1, 24);
             ClrEol;
             attribute := REVERSE;
          ELSE   (* exit Insert mode *)
             GetXY (cx, cy);
             GotoXY (1, 24);
             ClrEol;
             GotoXY (cx, cy);
             Insert := FALSE;
          END;
       END InsertMsg;


    PROCEDURE InsertOn;
    (* enter insert mode -- after INSERT MODE message is printed *)
       BEGIN
          attribute := NORMAL;
          GotoXY (cx, cy);
          Insert := TRUE;
       END InsertOn;


BEGIN   (* module initialization *)
   echo := Off;
   newline := FALSE;
   Insert := FALSE;
   EscState := FALSE;
   CurState1 := FALSE;
   CurState2 := FALSE;
END Term.
```

End Listing Ten

Listing Eleven

```
IMPLEMENTATION MODULE Screen;
(* module to perform "low level" screen functions (via AVIO) *)

   IMPORT ASCII;

   FROM SYSTEM IMPORT
      ADR;

   FROM Strings IMPORT
      Length;

   FROM Conversions IMPORT
      IntToString;

   FROM KH IMPORT
      IDM_GREEN;

   FROM Vio IMPORT
      VioSetCurPos, VioGetCurPos, VioScrollUp,
      VioWrtNCell, VioWrtTTY, VioCell;


   CONST
      GREY = 07H;
      WHITE = 0FH;
      REV_GY = 70H;
      GREEN = 02H;
      LITE_GRN = 0AH;
      REV_GRN = 20H;
      AMBER = 06H;
      LITE_AMB = 0EH;
      REV_AMB = 60H;
      RED = 0CH;
      CY_BK = 0B0H;
      CY_BL = 0B9H;
      REV_RD = 0CFH;
      REV_BL = 9FH;
      MAGENTA = 05H;


   VAR
      (* From Definition Module
      NORMAL : CARDINAL;
      HIGHLIGHT : CARDINAL;
      REVERSE : CARDINAL;
                attribute : CARDINAL;
      hvps : HVPS;
      *)
           x, y : CARDINAL;
           bCell : VioCell;
```

(continued on page 113)

Listing Eleven *(Listing continued, text begins on page 72.)*

```
1722:
1723:
1724:       PROCEDURE White;
1725:       (* Sets up colors: Monochrome White *)
1726:          BEGIN
1727:             NORMAL := GREY;
1728:             HIGHLIGHT := WHITE;
1729:             REVERSE := REV_GY;
1730:             attribute := NORMAL;
1731:          END White;
1732:
1733:
1734:       PROCEDURE Green;
1735:       (* Sets up colors: Monochrome Green *)
1736:          BEGIN
1737:             NORMAL := GREEN;
1738:             HIGHLIGHT := LITE_GRN;
1739:             REVERSE := REV_GRN;
1740:             attribute := NORMAL;
1741:          END Green;
1742:
1743:
1744:       PROCEDURE Amber;
1745:       (* Sets up colors: Monochrome Amber *)
1746:          BEGIN
1747:             NORMAL := AMBER;
1748:             HIGHLIGHT := LITE_AMB;
1749:             REVERSE := REV_AMB;
1750:             attribute := NORMAL;
1751:          END Amber;
1752:
1753:
1754:       PROCEDURE Color1;
1755:       (* Sets up colors: Blue, Red, Green *)
1756:          BEGIN
1757:             NORMAL := GREEN;
1758:             HIGHLIGHT := RED;
1759:             REVERSE := REV_BL;
1760:             attribute := NORMAL;
1761:          END Color1;
1762:
1763:
1764:       PROCEDURE Color2;
1765:       (* Sets up colors: Cyan Background; Black, Blue, White-on-Red *)
1766:          BEGIN
1767:             NORMAL := CY_BK;
1768:             HIGHLIGHT := CY_BL;
1769:             REVERSE := REV_RD;
1770:             attribute := NORMAL;
1771:          END Color2;
1772:
1773:
1774:       PROCEDURE HexToString (num : INTEGER;
1775:                              size : CARDINAL;
1776:                              VAR buf : ARRAY OF CHAR;
1777:                              VAR I : CARDINAL;
1778:                              VAR Done : BOOLEAN);
1779:    (* Local Procedure - convert number to string, represented in HEX *)
1780:
1781:          CONST
1782:             ZERO = 30H;    (* ASCII code *)
1783:             A = 41H;
1784:
1785:          VAR
1786:             i : CARDINAL;
1787:             h : CARDINAL;
1788:             t : ARRAY [0..10] OF CHAR;
1789:
1790:          BEGIN
1791:             i := 0;
1792:             REPEAT
1793:                h := num MOD 16;
1794:                IF h <= 9 THEN
1795:                   t[i] := CHR (h + ZERO);
1796:                ELSE
1797:                   t[i] := CHR (h - 10 + A);
1798:                END;
1799:                INC (i);
1800:                num := num DIV 16;
1801:             UNTIL num = 0;
1802:
1803:             IF (size > HIGH (buf)) OR (i > HIGH (buf)) THEN
1804:                Done := FALSE;
1805:                RETURN;
1806:             ELSE
1807:                Done := TRUE;
1808:             END;
1809:
1810:             WHILE size > i DO
1811:                buf[I] := '0';    (* pad with zeros *)
1812:                DEC (size);
1813:                INC (I);
1814:             END;
1815:
1816:             WHILE i > 0 DO
1817:                DEC (i);
1818:                buf[I] := t[i];
1819:                INC (I);
1820:             END;
1821:
1822:             buf[I] := 0C;
1823:          END HexToString;
1824:
1825:
1826:       PROCEDURE ClrScr;
1827:       (* Clear the screen, and home the cursor *)
```

(continued on page 114)

Listing Eleven *(Listing continued, text begins on page 72.)*

```
1828:          BEGIN
1829:             bCell.ch := ' ';          (* space = blank screen *)
1830:             bCell.attr := CHR (NORMAL); (* Normal Video Attribute *)
1831:             VioScrollUp (0, 0, 24, 79, 25, bCell, hvps);
1832:             GotoXY (0, 0);
1833:          END ClrScr;
1834:
1835:
1836:
1837:       PROCEDURE ClrEol;
1838:       (* clear from current cursor position to the end of the line *)
1839:          BEGIN
1840:             GetXY (x, y);      (* current cursor position *)
1841:             bCell.ch := ' ';     (* space = blank *)
1842:             bCell.attr := CHR (NORMAL);   (* Normal Video Attribute *)
1843:             VioScrollUp (y, x, y, 79, 1, bCell, hvps);
1844:          END ClrEol;
1845:
1846:
1847:       PROCEDURE Right;
1848:       (* move cursor to the right *)
1849:          BEGIN
1850:             GetXY (x, y);
1851:             INC (x);
1852:             GotoXY (x, y);
1853:          END Right;
1854:
1855:
1856:       PROCEDURE Left;
1857:       (* move cursor to the left *)
1858:          BEGIN
1859:             GetXY (x, y);
1860:             DEC (x);
1861:             GotoXY (x, y);
1862:          END Left;
1863:
1864:
1865:       PROCEDURE Up;
1866:       (* move cursor up *)
1867:          BEGIN
1868:             GetXY (x, y);
1869:             DEC (y);
1870:             GotoXY (x, y);
1871:          END Up;
1872:
1873:
1874:       PROCEDURE Down;
1875:       (* move cursor down *)
1876:          BEGIN
1877:             GetXY (x, y);
1878:             INC (y);
1879:             GotoXY (x, y);
1880:          END Down;
1881:
1882:
1883:       PROCEDURE GotoXY (col, row : CARDINAL);
1884:       (* position cursor at column, row *)
1885:          BEGIN
1886:             IF (col <= 79) AND (row <= 24) THEN
1887:                VioSetCurPos (row, col, hvps);
1888:             END;
1889:          END GotoXY;
1890:
1891:
1892:       PROCEDURE GetXY (VAR col, row : CARDINAL);
1893:       (* determine current cursor position *)
1894:          BEGIN
1895:             VioGetCurPos (row, col, hvps);
1896:          END GetXY;
1897:
1898:
1899:       PROCEDURE Write (c : CHAR);
1900:       (* Write a Character *)
1901:          BEGIN
1902:             WriteAtt (c);
1903:          END Write;
1904:
1905:
1906:       PROCEDURE WriteString (str : ARRAY OF CHAR);
1907:       (* Write String *)
1908:
1909:          VAR
1910:             i : CARDINAL;
1911:             c : CHAR;
1912:
1913:          BEGIN
1914:             i := 0;
1915:             c := str[i];
1916:             WHILE c # 0C DO
1917:                Write (c);
1918:                INC (i);
1919:                c := str[i];
1920:             END;
1921:          END WriteString;
1922:
1923:
1924:       PROCEDURE WriteInt (n : INTEGER; s : CARDINAL);
1925:       (* Write Integer *)
1926:
1927:          VAR
1928:             i : CARDINAL;
1929:             b : BOOLEAN;
1930:             str : ARRAY [0..6] OF CHAR;
1931:
1932:          BEGIN
1933:             i := 0;
1934:             IntToString (n, s, str, i, b);
1935:             WriteString (str);
1936:          END WriteInt;
1937:
1938:
1939:       PROCEDURE WriteHex (n, s : CARDINAL);
1940:       (* Write a Hexadecimal Number *)
1941:
1942:          VAR
1943:             i : CARDINAL;
1944:             b : BOOLEAN;
1945:             str : ARRAY [0..6] OF CHAR;
1946:
1947:          BEGIN
1948:             i := 0;
1949:             HexToString (n, s, str, i, b);
1950:             WriteString (str);
1951:          END WriteHex;
1952:
1953:
1954:       PROCEDURE WriteLn;
1955:       (* Write <cr> <lf> *)
1956:          BEGIN
1957:             Write (ASCII.cr);   Write (ASCII.lf);
1958:          END WriteLn;
1959:
1960:
1961:       PROCEDURE WriteAtt (c : CHAR);
1962:       (* write character and attribute at cursor position *)
1963:
1964:          VAR
1965:             s : ARRAY [0..1] OF CHAR;
1966:
1967:          BEGIN
1968:             GetXY (x, y);
1969:             IF (c = ASCII.ht) THEN
1970:                bCell.ch := ' ';
1971:                bCell.attr := CHR (attribute);
1972:                REPEAT
1973:                   VioWrtNCell (bCell, 1, y, x, hvps);
1974:                   Right;
1975:                UNTIL (x MOD 8) = 0;
1976:             ELSIF (c = ASCII.cr) OR (c = ASCII.lf)
1977:              OR (c = ASCII.bel) OR (c = ASCII.bs) THEN
1978:                s[0] := c;    s[1] := 0C;
1979:                VioWrtTTY (ADR (s), 1, hvps);
1980:                IF c = ASCII.lf THEN
1981:                   ClrEol;
1982:                END;
1983:             ELSE
1984:                bCell.ch := c;
1985:                bCell.attr := CHR (attribute);
1986:                VioWrtNCell (bCell, 1, y, x, hvps);
1987:                Right;
1988:             END;
1989:          END WriteAtt;
1990:
1991:    BEGIN     (* module initialization *)
1992:       ColorSet := IDM_GREEN;
1993:       NORMAL := GREEN;
1994:       HIGHLIGHT := LITE_GRN;
1995:       REVERSE := REV_GRN;
1996:       attribute := NORMAL;
1997:    END Screen.
1998:
```

End Listing Eleven

Listing Twelve

```
1999:    (*****************************************************************)
2000:    (*                                                               *)
2001:    (*                    Copyright (c) 1988, 1989                   *)
2002:    (*                     by Stony Brook Software                   *)
2003:    (*                             and                               *)
2004:    (*                       Copyright (c) 1990                      *)
2005:    (*                      by Brian R. Anderson                     *)
2006:    (*                       All rights reserved.                    *)
2007:    (*                                                               *)
2008:    (*****************************************************************)
2009:
2010:    IMPLEMENTATION MODULE CommPort [7];
2011:
2012:       FROM SYSTEM IMPORT
2013:          ADR, BYTE, WORD, ADDRESS;
2014:
2015:       FROM Storage IMPORT
2016:          ALLOCATE, DEALLOCATE;
2017:
2018:       FROM DosCalls IMPORT
2019:          DosOpen, AttributeSet, DosDevIOCtl, DosClose, DosRead, DosWrite;
2020:
2021:
2022:       TYPE
2023:          CP = POINTER TO CHAR;
2024:
2025:       VAR
2026:          pn : CARDINAL;
2027:          Handle : ARRAY [0..3] OF CARDINAL;
2028:          BufIn : ARRAY [0..3] OF CP;
2029:          BufOut : ARRAY [0..3] OF CP;
2030:          BufStart : ARRAY [0..3] OF CP;
2031:          BufLimit : ARRAY [0..3] OF CP;
2032:          BufSize : ARRAY [0..3] OF CARDINAL;
2033:          Temp : ARRAY [1..1024] OF CHAR; (* size of OS/2's serial queue *)
2034:
2035:
2036:       PROCEDURE CheckPort (portnum : CARDINAL) : BOOLEAN;
2037:       (* Check for valid port number and open port if not alredy open *)
2038:
2039:          CONST
2040:             PortName : ARRAY [0..3] OF ARRAY [0..4] OF CHAR =
2041:                [['COM1', 0C], ['COM2', 0C], ['COM3', 0C], ['COM4', 0C]];
2042:
```

(continued on page 116)

Listing Twelve *(Listing continued, text begins on page 72.)*

```
        VAR
           Action : CARDINAL;

        BEGIN
           (* check the port number *)
           IF portnum > 3 THEN
              RETURN FALSE;
           END;

           (* attempt to open the port if it is not already open *)
           IF Handle[portnum] = 0 THEN
           IF DosOpen(ADR(PortName[portnum]), Handle[portnum], Action, 0,
               AttributeSet{}, 1, 12H, 0) # 0 THEN
                 RETURN FALSE;
              END;
           END;
           RETURN TRUE;
        END CheckPort;


PROCEDURE InitPort (portnum : CARDINAL; speed : BaudRate; data :
                 DataBits; stop : StopBits; check : Parity) : CommStatus;
    (* Initialize a port *)

        CONST
           Rate : ARRAY BaudRate OF CARDINAL =
                    [110, 150, 300, 600, 1200, 2400, 4800, 9600, 19200];
           TransParity : ARRAY Parity OF BYTE = [2, 1, 0];

        TYPE
           LineChar =  RECORD
                          bDataBits : BYTE;
                          bParity : BYTE;
                          bStopBits : BYTE;
                       END;

        VAR
           LC : LineChar;

        BEGIN
           (* Check the port number *)
           IF NOT CheckPort(portnum) THEN
              RETURN InvalidPort;
           END;

           (* Set the baud rate *)
     IF DosDevIOCtl(0, ADR(Rate[speed]), 41H, 1, Handle[portnum]) # 0 THEN
              RETURN InvalidParameter;
           END;

           (* set the characteristics *)
           LC.bDataBits := BYTE(data);
           IF stop = 1 THEN
             DEC (stop);  (* 0x00 = 1 stop bits;    0x02 = 2 stop bits *)
           END;
           LC.bStopBits := BYTE(stop);
           LC.bParity := TransParity[check];

           IF DosDevIOCtl(0, ADR(LC), 42H, 1, Handle[portnum]) # 0 THEN
              RETURN InvalidParameter;
           END;

           RETURN Success;
        END InitPort;


     PROCEDURE StartReceiving (portnum, bufsize : CARDINAL) : CommStatus;
     (* Start receiving characters on a port *)
        BEGIN
           IF NOT CheckPort(portnum) THEN
              RETURN InvalidPort;
           END;
           IF BufStart[portnum] # NIL THEN
              RETURN AlreadyReceiving;
           END;
           ALLOCATE (BufStart[portnum], bufsize);
           BufIn[portnum] := BufStart[portnum];
           BufOut[portnum] := BufStart[portnum];
           BufLimit[portnum] := BufStart[portnum];
           INC (BufLimit[portnum]:ADDRESS, bufsize - 1);
           BufSize[portnum] := bufsize;
           RETURN Success;
        END StartReceiving;


     PROCEDURE StopReceiving (portnum : CARDINAL) : CommStatus;
     (* Stop receiving characters on a port *)
        BEGIN
           IF NOT CheckPort(portnum) THEN
              RETURN InvalidPort;
           END;
           IF BufStart[portnum] # NIL THEN
              DEALLOCATE (BufStart[portnum], BufSize[portnum]);
              BufLimit[portnum] := NIL;
              BufIn[portnum] := NIL;
              BufOut[portnum] := NIL;
              BufSize[portnum] := 0;
           END;
           DosClose(Handle[portnum]);
           Handle[portnum] := 0;
           RETURN Success;
        END StopReceiving;


     PROCEDURE GetChar (portnum : CARDINAL; VAR ch : CHAR) : CommStatus;
     (* Get a character from the comm port *)

        VAR
           status : CARDINAL;
           read : CARDINAL;
           que : RECORD
                    ct : CARDINAL;
                    sz : CARDINAL;
                 END;
           i : CARDINAL;

        BEGIN
           IF BufStart[portnum] = NIL THEN
              RETURN NotReceiving;
           END;
           IF NOT CheckPort(portnum) THEN
              RETURN InvalidPort;
           END;
           status := DosDevIOCtl (ADR (que), 0, 68H, 1, Handle[portnum]);
           IF (status = 0) AND (que.ct # 0) THEN
           status := DosRead (Handle[portnum], ADR (Temp), que.ct, read);
              IF (status # 0) OR (read = 0) THEN
                 RETURN NotReceiving;
              END;
              FOR i := 1 TO read DO
                 BufIn[portnum]^ := Temp[i];
                 IF BufIn[portnum] = BufLimit[portnum] THEN
                    BufIn[portnum] := BufStart[portnum];
                 ELSE
                    INC (BufIn[portnum]:ADDRESS);
                 END;
                 IF BufIn[portnum] = BufOut[portnum] THEN
                    RETURN BufferOverflow;
                 END;
              END;
           END;

           IF BufIn[portnum] = BufOut[portnum] THEN
              RETURN NoCharacter;
           END;
           ch := BufOut[portnum]^;
           IF BufOut[portnum] = BufLimit[portnum] THEN
              BufOut[portnum] := BufStart[portnum];
           ELSE
              INC (BufOut[portnum]:ADDRESS);
           END;
           RETURN Success;
        END GetChar;


     PROCEDURE SendChar (portnum : CARDINAL; ch : CHAR;
                         modem : BOOLEAN) : CommStatus;
     (* send a character to the comm port *)

        VAR
           wrote : CARDINAL;
           status : CARDINAL;
           commSt : CHAR;

        BEGIN
           IF NOT CheckPort(portnum) THEN
              RETURN InvalidPort;
           END;
        status := DosDevIOCtl (ADR (commSt), 0, 64H, 1, Handle[portnum]);
           IF (status # 0) OR (commSt # 0C) THEN
              RETURN TimeOut;
           ELSE
              status := DosWrite(Handle[portnum], ADR(ch), 1, wrote);
              IF (status # 0) OR (wrote # 1) THEN
                 RETURN TimeOut;
              ELSE
                 RETURN Success;
              END;
           END;
        END SendChar;


   BEGIN   (* module initialization *)
      (* nothing open yet *)
      FOR pn := 0 TO 3 DO
         Handle[pn] := 0;
         BufStart[pn] := NIL;
         BufLimit[pn] := NIL;
         BufIn[pn] := NIL;
         BufOut[pn] := NIL;
         BufSize[pn] := 0;
      END;
   END CommPort.
```

End Listing Twelve

Listing Thirteen

```
IMPLEMENTATION MODULE Files;   (* File I/O for Kermit *)

   FROM FileSystem IMPORT
      File, Response, Delete, Lookup, Close, ReadNBytes, WriteNBytes;
   FROM Strings IMPORT
      Append;
   FROM Conversions IMPORT
      CardToString;
   FROM SYSTEM IMPORT
      ADR, SIZE;
   TYPE
      buffer = ARRAY [1..512] OF CHAR;
   VAR
      ext : CARDINAL;  (* new file extensions to avoid name conflict *)
      inBuf, outBuf : buffer;
      inP, outP : CARDINAL;   (* buffer pointers *)
      read, written : CARDINAL;   (* number of bytes read or written *)
                                  (* by ReadNBytes or WriteNBytes    *)
   PROCEDURE Open (VAR f : File; name : ARRAY OF CHAR) : Status;
   (* opens an existing file for reading, returns status *)
```

```
   BEGIN
      Lookup (f, name, FALSE);
      IF f.res = done THEN
         inP := 0;   read := 0;
         RETURN Done;
      ELSE
         RETURN Error;
      END;
   END Open;
PROCEDURE Create (VAR f : File; name : ARRAY OF CHAR) : Status;
(* creates a new file for writing, returns status *)
   VAR
      ch : CHAR;
      str : ARRAY [0..3] OF CHAR;
      i : CARDINAL;
      b : BOOLEAN;
   BEGIN
      LOOP
         Lookup (f, name, FALSE);   (* check to see if file exists *)
         IF f.res = done THEN
            Close (f);
            (* Filename Clash: Change file name *)
            IF ext > 99 THEN   (* out of new names... *)
               RETURN Error;
            END;
            i := 0;
            WHILE (name[i] # 0C) AND (name[i] # '.') DO
               INC (i);   (* scan for end of filename *)
            END;
            name[i] := '.';   name[i + 1] := 'K';   name[i + 2] := 0C;
            i := 0;
            CardToString (ext, 1, str, i, b);
               Append (name, str);   (* append new extension *)
               INC (ext);
            ELSE
               EXIT;
            END;
         END;
         Lookup (f, name, TRUE);
         IF f.res = done THEN
            outP := 0;
            RETURN Done;
         ELSE
            RETURN Error;
         END;
      END Create;
   PROCEDURE CloseFile (VAR f : File; Which : FileType) : Status;
   (* closes a file after reading or writing *)
      BEGIN
         written := outP;
         IF (Which = Output) AND (outP > 0) THEN
            WriteNBytes (f, ADR (outBuf), outP);
            written := f.count;
         END;
         Close (f);
         IF (written = outP) AND (f.res = done) THEN
            RETURN Done;
         ELSE
            RETURN Error;
         END;
      END CloseFile;
   PROCEDURE Get (VAR f : File; VAR ch : CHAR) : Status;
   (* Reads one character from the file, returns status *)
      BEGIN
         IF inP = read THEN
            ReadNBytes (f, ADR (inBuf), SIZE (inBuf));
            read := f.count;
            inP := 0;
         END;
         IF read = 0 THEN
            RETURN EOF;
         ELSE
            INC (inP);
            ch := inBuf[inP];
            RETURN Done;
         END;
      END Get;
   PROCEDURE Put (ch : CHAR);
   (* Writes one character to the file buffer *)
      BEGIN
         INC (outP);
         outBuf[outP] := ch;
      END Put;
   PROCEDURE DoWrite (VAR f : File) : Status;
   (* Writes buffer to disk only if nearly full *)
      BEGIN
         IF outP < 400 THEN   (* still room in buffer *)
            RETURN Done;
         ELSE
            WriteNBytes (f, ADR (outBuf), outP);
            written := f.count;
            IF (written = outP) AND (f.res = done) THEN
               outP := 0;
               RETURN Done;
            ELSE
               RETURN Error;
            END;
         END;
      END DoWrite;
BEGIN (* module initialization *)
   ext := 0;
END Files.
```

End Listing Thirteen

Listing Fourteen

```
DEFINITION MODULE KH;

CONST
   ID_OK        =  25;
```

(continued on page 118)

Listing Fourteen *(Listing continued, text begins on page 72.)*

```
   PARITY_OFF    =  150;
   ID_NONE       =  152;
   ID_ODD        =  151;
   ID_EVEN       =  150;

   STOP_OFF      =  140;
   ID_STOP2      =  142;
   ID_STOP1      =  141;

   DATA_OFF      =  130;
   ID_DATA8      =  138;
   ID_DATA7      =  137;

   BAUD_OFF      =  120;
   ID_B19K2      =  128;
   ID_B9600      =  127;
   ID_B4800      =  126;
   ID_B2400      =  125;
   ID_B1200      =  124;
   ID_B600       =  123;
   ID_B300       =  122;
   ID_B150       =  121;
   ID_B110       =  120;

   COM_OFF       =  100;
   ID_COM2       =  101;
   ID_COM1       =  100;

   IDM_C2        =  24;
   IDM_C1        =  23;
   IDM_AMBER     =  22;
   IDM_GREEN     =  21;
   IDM_WHITE     =  20;
   IDM_COLORS    =  19;
   IDM_DIREND    =  18;
   ID_DIRPATH    =  17;
   ID_SENDFN     =  16;
   IDM_DIRPATH   =  15;
   IDM_SENDFN    =  14;
   IDM_TERMHELP  =  13;
   IDM_HELPMENU  =  12;
   IDM_ABOUT     =  11;
   IDM_PARITY    =  10;
   IDM_STOPBITS  =  9;
   IDM_DATABITS  =  8;
   IDM_BAUDRATE  =  7;
   IDM_COMPORT   =  6;
   IDM_QUIT      =  5;
   IDM_REC       =  4;
   IDM_SEND      =  3;
   IDM_CONNECT   =  2;
   IDM_DIR       =  1;
   IDM_OPTIONS   =  52;
   IDM_FILE      =  51;
   IDM_KERMIT    =  50;

END KH.
```

End Listing Fourteen

Listing Fifteen

```
IMPLEMENTATION MODULE KH;
END KH.
```

End Listing Fifteen

Listing Sixteen

```
#define IDM_KERMIT      50
#define IDM_FILE        51
#define IDM_OPTIONS     52
#define IDM_HELP        0
#define IDM_DIR         1
#define IDM_CONNECT     2
#define IDM_SEND        3
#define IDM_REC         4
#define IDM_QUIT        5
#define IDM_COMPORT     6
#define IDM_BAUDRATE    7
#define IDM_DATABITS    8
#define IDM_STOPBITS    9
#define IDM_PARITY      10
#define IDM_ABOUT       11
#define IDM_HELPMENU    12
#define IDM_TERMHELP    13
#define IDM_SENDFN      14
#define IDM_DIRPATH     15
#define ID_SENDFN       16
#define ID_DIRPATH      17
#define IDM_DIREND      18
#define IDM_COLORS      19
#define IDM_WHITE       20
#define IDM_GREEN       21
#define IDM_AMBER       22
#define IDM_C1          23
#define IDM_C2          24
#define ID_OK           25
#define ID_COM1         100
#define ID_COM2         101
#define ID_B110         120
#define ID_B150         121
#define ID_B300         122
#define ID_B600         123
#define ID_B1200        124
#define ID_B2400        125
#define ID_B4800        126
#define ID_B9600        127
#define ID_B19K2        128
#define ID_DATA7        137
#define ID_DATA8        138
#define ID_STOP1        141
#define ID_STOP2        142
#define ID_EVEN         150
#define ID_ODD          151
#define ID_NONE         152
```

End Listing Sixteen

Listing Seventeen

```
IMPLEMENTATION MODULE DataLink;  (* Sends and Receives Packets for PCKermit *)

   FROM ElapsedTime IMPORT
      StartTime, GetTime;
   FROM Screen IMPORT
      ClrScr, WriteString, WriteLn;
   FROM OS2DEF IMPORT
      HIWORD, LOWORD;
   FROM PMWIN IMPORT
      MPARAM, MPFROM2SHORT, WinPostMsg;
   FROM Shell IMPORT
      ChildFrameWindow, comport;
   FROM CommPort IMPORT
      CommStatus, GetChar, SendChar;
   FROM PAD IMPORT
      PacketType, yourNPAD, yourPADC, yourEOL;
   FROM KH IMPORT
      COM_OFF;
   FROM SYSTEM IMPORT
      BYTE;
   IMPORT ASCII;
   CONST
      MAXtime = 100;   (* hundredths of a second -- i.e., one second *)
      MAXsohtrys = 100;
      DL_BadCS = 1;
      DL_NoSOH = 2;
   TYPE
      SMALLSET = SET OF [0..7];   (* BYTE *)
   VAR
      ch : CHAR;
      status : CommStatus;
   PROCEDURE Delay (t : CARDINAL);
   (* delay time in milliseconds *)
      VAR
         tmp : LONGINT;
      BEGIN
         tmp := t DIV 10;
         StartTime;
         WHILE GetTime() < tmp DO
```

```
      END;
   END Delay;
PROCEDURE ByteAnd (a, b : BYTE) : BYTE;
   BEGIN
      RETURN BYTE (SMALLSET (a) * SMALLSET (b));
   END ByteAnd;
PROCEDURE Char (c : INTEGER) : CHAR;
(* converts a number 0-95 into a printable character *)
   BEGIN
      RETURN (CHR (CARDINAL (ABS (c) + 32)));
   END Char;
PROCEDURE UnChar (c : CHAR) : INTEGER;
(* converts a character into its corresponding number *)
   BEGIN
      RETURN (ABS (INTEGER (ORD (c)) - 32));
   END UnChar;
PROCEDURE FlushUART;
(* ensure no characters left in UART holding registers *)
   BEGIN
      Delay (500);
      REPEAT
         status := GetChar (comport - COM_OFF, ch);
      UNTIL status = NoCharacter;
   END FlushUART;
PROCEDURE SendPacket (s : PacketType);
(* Adds SOH and CheckSum to packet *)
   VAR
      i : CARDINAL;
      checksum : INTEGER;
   BEGIN
      Delay (10);   (* give host a chance to catch its breath *)
      FOR i := 1 TO yourNPAD DO
         status := SendChar (comport - COM_OFF, yourPADC, FALSE);
      END;
      status := SendChar (comport - COM_OFF, ASCII.soh, FALSE);
      i := 1;
      checksum := 0;
      WHILE s[i] # 0C DO
         INC (checksum, ORD (s[i]));
         status := SendChar (comport - COM_OFF, s[i], FALSE);
         INC (i);
      END;
      checksum := checksum + (INTEGER (BITSET (checksum) * {7, 6}) DIV 64);
      checksum := INTEGER (BITSET (checksum) * {5, 4, 3, 2, 1, 0});
      status := SendChar (comport - COM_OFF, Char (checksum), FALSE);
      IF yourEOL # 0C THEN
         status := SendChar (comport - COM_OFF, yourEOL, FALSE);
      END;
   END SendPacket;
PROCEDURE ReceivePacket (VAR r : PacketType) : BOOLEAN;
(* strips SOH and checksum -- returns status: TRUE = good packet     *)
(* received;  FALSE = timed out waiting for packet or checksum error *)
   VAR
      sohtrys : INTEGER;
      i, len : INTEGER;
      ch : CHAR;
      checksum : INTEGER;
      mycheck, yourcheck : CHAR;
   BEGIN
      sohtrys := MAXsohtrys;
      REPEAT
         StartTime;
         REPEAT
            status := GetChar (comport - COM_OFF, ch);
         UNTIL (status = Success) OR (GetTime() > MAXtime);
         ch := CHAR (ByteAnd (ch, 177C));   (* mask off MSB *)
         (* skip over up to MAXsohtrys padding characters, *)
         (* but allow only MAXsohtrys/10 timeouts          *)
         IF status = Success THEN
            DEC (sohtrys);
         ELSE
            DEC (sohtrys, 10);
         END;
      UNTIL (ch = ASCII.soh) OR (sohtrys <= 0);
      IF ch = ASCII.soh THEN
         (* receive rest of packet *)
         StartTime;
         REPEAT
            status := GetChar (comport - COM_OFF, ch);
         UNTIL (status = Success) OR (GetTime() > MAXtime);
         ch := CHAR (ByteAnd (ch, 177C));
         len := UnChar (ch);
         r[1] := ch;
         checksum := ORD (ch);
         i := 2;   (* on to second character in packet -- after LEN *)
         REPEAT
            StartTime;
            REPEAT
               status := GetChar (comport - COM_OFF, ch);
            UNTIL (status = Success) OR (GetTime() > MAXtime);
            ch := CHAR (ByteAnd (ch, 177C));
            r[i] := ch;   INC (i);
            INC (checksum, (ORD (ch)));
         UNTIL (i > len);
         (* get checksum character *)
         StartTime;
         REPEAT
            status := GetChar (comport - COM_OFF, ch);
         UNTIL (status = Success) OR (GetTime() > MAXtime);
         ch := CHAR (ByteAnd (ch, 177C));
         yourcheck := ch;
         r[i] := 0C;
         checksum := checksum +
                        (INTEGER (BITSET (checksum) * {7, 6}) DIV 64);
         checksum := INTEGER (BITSET (checksum) *  {5, 4, 3, 2, 1, 0});
         mycheck := Char (checksum);
         IF mycheck = yourcheck THEN    (* checksum OK *)
            RETURN TRUE;
         ELSE   (* ERROR!!! *)
            WinPostMsg (ChildFrameWindow, WM_DL,
                        MPFROM2SHORT (DL_BadCS, 0), 0);
            RETURN FALSE;
         END;
      ELSE
         WinPostMsg (ChildFrameWindow, WM_DL,
                        MPFROM2SHORT (DL_NoSOH, 0), 0);
         RETURN FALSE;
      END;
   END ReceivePacket;
PROCEDURE DoDLMsg (mp1, mp2 : MPARAM);
(* Process DataLink Messages *)
   BEGIN
      CASE LOWORD (mp1) OF
         DL_BadCS:
            WriteString ("Bad Checksum");   WriteLn;
       | DL_NoSOH:
            WriteString ("No SOH");   WriteLn;
      ELSE
         (* Do Nothing *)
      END;
   END DoDLMsg;
END DataLink.
```

End Listing Seventeen

Listing Eighteen

```
#include <os2.h>
#include "pckermit.h"

ICON IDM_KERMIT pckermit.ico

MENU IDM_KERMIT
   BEGIN
      SUBMENU "~File", IDM_FILE
         BEGIN
            MENUITEM "~Directory...",     IDM_DIR
            MENUITEM "~Connect\t^C",      IDM_CONNECT
            MENUITEM "~Send...\t^S",      IDM_SEND
            MENUITEM "~Receive...\t^R",   IDM_REC
            MENUITEM SEPARATOR
            MENUITEM "E~xit\t^X",         IDM_QUIT
            MENUITEM "A~bout PCKermit...", IDM_ABOUT
         END

      SUBMENU "~Options", IDM_OPTIONS
         BEGIN
            MENUITEM "~COM port...",      IDM_COMPORT
            MENUITEM "~Baud rate...",     IDM_BAUDRATE
            MENUITEM "~Data bits...",     IDM_DATABITS
            MENUITEM "~Stop bits...",     IDM_STOPBITS
            MENUITEM "~Parity bits...",   IDM_PARITY
         END
```

(continued on page 120)

Listing Eighteen *(Listing continued, text begins on page 72.)*

```
2264:
2265:         SUBMENU "~Colors", IDM_COLORS
2266:            BEGIN
2267:               MENUITEM "~White Mono",        IDM_WHITE
2268:               MENUITEM "~Green Mono",        IDM_GREEN
2269:               MENUITEM "~Amber Mono",        IDM_AMBER
2270:               MENUITEM "Full Color ~1",      IDM_C1
2271:               MENUITEM "Full Color ~2",      IDM_C2
2272:            END
2273:
2274:         MENUITEM "F1=Help",    IDM_HELP, MIS_HELP | MIS_BUTTONSEPARATOR
2275:      END
2276:
2277:   ACCELTABLE IDM_KERMIT
2278:      BEGIN
2279:         "^C", IDM_CONNECT
2280:         "^S", IDM_SEND
2281:         "^R", IDM_REC
2282:         "^X", IDM_QUIT
2283:      END
2284:
2285:   DLGTEMPLATE IDM_COMPORT LOADONCALL MOVEABLE DISCARDABLE
2286:   BEGIN
2287:      DIALOG "", IDM_COMPORT, 129, 91, 143, 54, FS_NOBYTEALIGN |
2288:         FS_DLGBORDER | WS_VISIBLE | WS_CLIPSIBLINGS | WS_SAVEBITS
2289:       BEGIN
2290:          CONTROL "Select COM Port", IDM_COMPORT, 10, 9, 83, 38,
2291:                  WC_STATIC, SS_GROUPBOX | WS_VISIBLE
2292:          CONTROL "COM1", ID_COM1, 30, 25, 43, 10, WC_BUTTON,
2293:                  BS_AUTORADIOBUTTON | WS_GROUP | WS_TABSTOP | WS_VISIBLE
2294:          CONTROL "COM2", ID_COM2, 30, 15, 39, 10, WC_BUTTON,
2295:                  BS_AUTORADIOBUTTON | WS_TABSTOP | WS_VISIBLE
2296:          CONTROL "OK", ID_OK, 101, 10, 38, 12, WC_BUTTON,
2297:          BS_PUSHBUTTON | BS_DEFAULT | WS_GROUP | WS_TABSTOP | WS_VISIBLE
2298:       END
2299:   END
2300:
2301:   DLGTEMPLATE IDM_BAUDRATE LOADONCALL MOVEABLE DISCARDABLE
2302:   BEGIN
2303:       DIALOG "", IDM_BAUDRATE, 131, 54, 142, 115, FS_NOBYTEALIGN |
2304:              FS_DLGBORDER | WS_VISIBLE | WS_CLIPSIBLINGS | WS_SAVEBITS
2305:       BEGIN
2306:          CONTROL "Select Baud Rate", IDM_BAUDRATE, 8, 6, 85, 107,
2307:                  WC_STATIC, SS_GROUPBOX | WS_VISIBLE
2308:          CONTROL "110 Baud", ID_B110, 20, 90, 62, 10, WC_BUTTON,
2309:                  BS_AUTORADIOBUTTON | WS_GROUP | WS_TABSTOP | WS_VISIBLE
2310:          CONTROL "150 Baud", ID_B150, 20, 80, 57, 10, WC_BUTTON,
2311:                  BS_AUTORADIOBUTTON | WS_TABSTOP | WS_VISIBLE
2312:          CONTROL "300 Baud", ID_B300, 20, 70, 58, 10, WC_BUTTON,
2313:                  BS_AUTORADIOBUTTON | WS_TABSTOP | WS_VISIBLE
2314:          CONTROL "600 Baud", ID_B600, 20, 60, 54, 10, WC_BUTTON,
2315:                  BS_AUTORADIOBUTTON | WS_TABSTOP | WS_VISIBLE
2316:          CONTROL "1200 Baud", ID_B1200, 20, 50, 59, 10, WC_BUTTON,
2317:                  BS_AUTORADIOBUTTON | WS_TABSTOP | WS_VISIBLE
2318:          CONTROL "2400 Baud", ID_B2400, 20, 40, 63, 10, WC_BUTTON,
2319:                  BS_AUTORADIOBUTTON | WS_TABSTOP | WS_VISIBLE
2320:          CONTROL "4800 Baud", ID_B4800, 20, 30, 62, 10, WC_BUTTON,
2321:                  BS_AUTORADIOBUTTON | WS_TABSTOP | WS_VISIBLE
2322:          CONTROL "9600 Baud", ID_B9600, 20, 20, 59, 10, WC_BUTTON,
2323:                  BS_AUTORADIOBUTTON | WS_TABSTOP | WS_VISIBLE
2324:          CONTROL "19,200 Baud", ID_B19K2, 20, 10, 69, 10, WC_BUTTON,
2325:                  BS_AUTORADIOBUTTON | WS_TABSTOP | WS_VISIBLE
2326:          CONTROL "OK", ID_OK, 100, 8, 38, 12, WC_BUTTON, BS_PUSHBUTTON |
2327:                  BS_DEFAULT | WS_GROUP | WS_TABSTOP | WS_VISIBLE
2328:       END
2329:   END
2330:
2331:   DLGTEMPLATE IDM_DATABITS LOADONCALL MOVEABLE DISCARDABLE
2332:   BEGIN
2333:       DIALOG "", IDM_DATABITS, 137, 80, 140, 56, FS_NOBYTEALIGN |
2334:                  FS_DLGBORDER | WS_VISIBLE | WS_SAVEBITS
2335:       BEGIN
2336:          CONTROL "Select Data Bits", IDM_DATABITS, 8, 11, 80, 36,
2337:                  WC_STATIC, SS_GROUPBOX | WS_VISIBLE
2338:          CONTROL "7 Data Bits", ID_DATA7, 15, 25, 67, 10, WC_BUTTON,
2339:                  BS_AUTORADIOBUTTON | WS_GROUP | WS_TABSTOP | WS_VISIBLE
2340:          CONTROL "8 Data Bits", ID_DATA8, 15, 15, 64, 10, WC_BUTTON,
2341:                  BS_AUTORADIOBUTTON | WS_TABSTOP | WS_VISIBLE
2342:          CONTROL "OK", ID_OK, 96, 12, 38, 12, WC_BUTTON, BS_PUSHBUTTON |
2343:                  BS_DEFAULT | WS_GROUP | WS_TABSTOP | WS_VISIBLE
2344:       END
2345:   END
2346:
2347:   DLGTEMPLATE IDM_STOPBITS LOADONCALL MOVEABLE DISCARDABLE
2348:   BEGIN
2349:       DIALOG "", IDM_STOPBITS, 139, 92, 140, 43, FS_NOBYTEALIGN |
2350:                  FS_DLGBORDER | WS_VISIBLE | WS_SAVEBITS
2351:       BEGIN
2352:          CONTROL "Select Stop Bits", IDM_STOPBITS, 9, 6, 80, 32,
2353:                  WC_STATIC, SS_GROUPBOX | WS_VISIBLE
2354:          CONTROL "1 Stop Bit", ID_STOP1, 20, 20, 57, 10, WC_BUTTON,
2355:                  BS_AUTORADIOBUTTON | WS_GROUP | WS_TABSTOP | WS_VISIBLE
2356:          CONTROL "2 Stop Bits", ID_STOP2, 20, 10, 60, 10, WC_BUTTON,
2357:                  BS_AUTORADIOBUTTON | WS_TABSTOP | WS_VISIBLE
2358:          CONTROL "OK", ID_OK, 96, 8, 38, 12, WC_BUTTON, BS_PUSHBUTTON |
2359:                  BS_DEFAULT | WS_GROUP | WS_TABSTOP | WS_VISIBLE
2360:       END
2361:   END
2362:
2363:   DLGTEMPLATE IDM_PARITY LOADONCALL MOVEABLE DISCARDABLE
2364:   BEGIN
2365:      DIALOG "", IDM_PARITY, 138, 84, 134, 57, FS_NOBYTEALIGN |
2366:            FS_DLGBORDER | WS_VISIBLE | WS_SAVEBITS
2367:       BEGIN
2368:          CONTROL "Select Parity", IDM_PARITY, 12, 6, 64, 46, WC_STATIC,
2369:          SS_GROUPBOX | WS_VISIBLE
2370:          CONTROL "Even", ID_EVEN, 25, 30, 40, 10, WC_BUTTON,
2371:                  BS_AUTORADIOBUTTON | WS_GROUP | WS_TABSTOP | WS_VISIBLE
2372:          CONTROL "Odd", ID_ODD, 25, 20, 38, 10, WC_BUTTON,
```

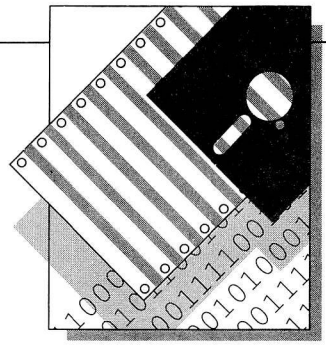
```
                BS_AUTORADIOBUTTON | WS_TABSTOP | WS_VISIBLE
          CONTROL "None", ID_NONE, 25, 10, 40, 10, WC_BUTTON,
                BS_AUTORADIOBUTTON | WS_TABSTOP | WS_VISIBLE
          CONTROL "OK", ID_OK, 88, 8, 38, 12, WC_BUTTON, BS_PUSHBUTTON |
                BS_DEFAULT | WS_GROUP | WS_TABSTOP | WS_VISIBLE
      END
  END


  DLGTEMPLATE IDM_ABOUT LOADONCALL MOVEABLE DISCARDABLE
  BEGIN
      DIALOG "", IDM_ABOUT, 93, 74, 229, 88, FS_NOBYTEALIGN |
           FS_DLGBORDER | WS_VISIBLE | WS_SAVEBITS
      BEGIN
          ICON IDM_KERMIT -1, 12, 64, 22, 16
CONTROL "PCKermit for OS/2", 256, 67, 70, 82, 8, WC_STATIC,
         SS_TEXT | DT_LEFT | DT_TOP | WS_GROUP | WS_VISIBLE
CONTROL "Copyright (c) 1990 by Brian R. Anderson", 257, 27, 30, 172, 8,
         WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_GROUP | WS_VISIBLE
CONTROL "Microcomputer to Mainframe Communications", 259, 13, 50, 199, 8,
         WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_GROUP | WS_VISIBLE
CONTROL "  OK  ", 258, 88, 10, 38, 12, WC_BUTTON, BS_PUSHBUTTON |
                BS_DEFAULT | WS_TABSTOP | WS_VISIBLE
END
END


  DLGTEMPLATE IDM_HELPMENU LOADONCALL MOVEABLE DISCARDABLE
  BEGIN
      DIALOG "", IDM_HELPMENU, 83, 45, 224, 125, FS_NOBYTEALIGN |
           FS_DLGBORDER | WS_VISIBLE | WS_CLIPSIBLINGS | WS_SAVEBITS
      BEGIN
        ICON IDM_KERMIT -1, 14, 99, 21, 16
        CONTROL "PCKermit Help Menu", 256, 64, 106, 91, 8, WC_STATIC,
                SS_TEXT | DT_LEFT | DT_TOP | WS_GROUP | WS_VISIBLE
        CONTROL "set communications Options ..... Alt, O",
            258, 10, 80, 201, 8, WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP |
                WS_GROUP | WS_VISIBLE
        CONTROL "Connect to Host ..... Alt, F; C",
            259, 10, 70, 204, 8, WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP |
            WS_GROUP | WS_VISIBLE
        CONTROL "Directory ..... Alt, F; D",
            260, 10, 60, 207, 8, WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP |
            WS_GROUP | WS_VISIBLE
        CONTROL "Send a File ..... Alt, F; S",
            261, 10, 50, 207, 8, WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP |
            WS_GROUP | WS_VISIBLE
        CONTROL "Receive a File ..... Alt, F; R",
            262, 10, 40, 209, 8, WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP |
            WS_GROUP | WS_VISIBLE
        CONTROL "Exit ..... Alt, F; X",
            263, 10, 30, 205, 8, WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP |
            WS_GROUP | WS_VISIBLE
        CONTROL "OK", 264, 83, 9, 38, 12, WC_BUTTON, BS_PUSHBUTTON |
            WS_TABSTOP | WS_VISIBLE | BS_DEFAULT
      END
  END


  DLGTEMPLATE IDM_TERMHELP LOADONCALL MOVEABLE DISCARDABLE
  BEGIN
      DIALOG "", IDM_TERMHELP, 81, 20, 238, 177, FS_NOBYTEALIGN |
           FS_DLGBORDER | WS_VISIBLE | WS_CLIPSIBLINGS | WS_SAVEBITS
  BEGIN
  CONTROL "^E = Echo mode", 256, 10, 160, 72, 8, WC_STATIC,
          SS_TEXT | DT_LEFT | DT_TOP | WS_GROUP | WS_VISIBLE
  CONTROL "^L = Local echo mode", 257, 10, 150, 97, 8, WC_STATIC,
          SS_TEXT | DT_LEFT | DT_TOP | WS_GROUP | WS_VISIBLE
  CONTROL "^T = Terminal Mode (no echo)", 258, 10, 140, 131, 8, WC_STATIC,
          SS_TEXT | DT_LEFT | DT_TOP | WS_GROUP | WS_VISIBLE
  CONTROL "^N = Newline mode (<cr> --> <cr><lf>)", 259, 10, 130, 165, 8,
          WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_GROUP | WS_VISIBLE
  CONTROL "^O = Newline mode OFF", 260, 10, 120, 109, 8, WC_STATIC,
          SS_TEXT | DT_LEFT | DT_TOP | WS_GROUP | WS_VISIBLE
  CONTROL "Televideo TVI950/IBM 7171 Term. Emul.", 261, 10, 100, 217, 8,
          WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_GROUP | WS_VISIBLE
  CONTROL "Sh-F1 - Sh-F12   =   PF1 - PF12", 262, 10, 90, 135, 8,
          WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_GROUP | WS_VISIBLE
  CONTROL "Home                 =  Clear", 263, 10, 80, 119, 8, WC_STATIC,
          SS_TEXT | DT_LEFT | DT_TOP | WS_GROUP | WS_VISIBLE
  CONTROL "PgDn                 =  Page  Down (as used in PROFS)",
           264, 10, 70, 228, 8, WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP |
           WS_GROUP | WS_VISIBLE
  CONTROL "PgUp                 =  Page Up (as used in PROFS)",
           265, 10, 60, 227, 8, WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP |
           WS_GROUP | WS_VISIBLE
  CONTROL "Insert       =  Insert (Enter to Clear)", 266, 10, 40, 221, 8,
           WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_GROUP | WS_VISIBLE
  CONTROL "Delete               =  Delete", 267, 10, 30, 199, 8,
           WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_GROUP | WS_VISIBLE
  CONTROL "Control-G  = Reset (rewrites the screen)", 268, 10, 20, 222, 8,
          WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_GROUP | WS_VISIBLE
  CONTROL "Cursor Keys (i.e., Up, Down, Left, Right) all work.",
          269, 10, 10, 229, 8, WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP |
          WS_GROUP | WS_VISIBLE
  CONTROL "OK", 270, 193, 158, 38, 12, WC_BUTTON, BS_PUSHBUTTON |
          BS_DEFAULT | WS_TABSTOP | WS_VISIBLE
  CONTROL "End          =  End (as used in PROFS)", 271, 10, 50, 209, 8,
          WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP | WS_GROUP | WS_VISIBLE
  END
  END


  DLGTEMPLATE IDM_SENDFN LOADONCALL MOVEABLE DISCARDABLE
  BEGIN
  DIALOG "", IDM_SENDFN, 113, 90, 202, 60, FS_NOBYTEALIGN | FS_DLGBORDER |
                  WS_VISIBLE | WS_SAVEBITS
   BEGIN
      CONTROL "Send File", 256, 4, 4, 195, 24, WC_STATIC, SS_GROUPBOX |
           WS_GROUP | WS_VISIBLE
      CONTROL "Enter filename:", 257, 13, 11, 69, 8, WC_STATIC, SS_TEXT |
           DT_LEFT | DT_TOP | WS_GROUP | WS_VISIBLE
      ICON IDM_KERMIT -1, 15, 38, 22, 16
      CONTROL "PCKermit for OS/2", 259, 59, 45, 82, 8, WC_STATIC,
           SS_TEXT | DT_LEFT | DT_TOP | WS_GROUP | WS_VISIBLE
      CONTROL "OK", 260, 154, 36, 38, 12, WC_BUTTON, BS_PUSHBUTTON |
           WS_TABSTOP | WS_VISIBLE | BS_DEFAULT
      CONTROL "", ID_SENDFN, 89, 10, 98, 8, WC_ENTRYFIELD, ES_LEFT |
           ES_MARGIN | WS_TABSTOP | WS_VISIBLE
  END
  END


  DLGTEMPLATE IDM_DIRPATH LOADONCALL MOVEABLE DISCARDABLE
  BEGIN
  DIALOG "", IDM_DIRPATH, 83, 95, 242, 46, FS_NOBYTEALIGN |
          FS_DLGBORDER | WS_VISIBLE | WS_SAVEBITS
  BEGIN
       CONTROL "Directory", 256, 7, 5, 227, 24, WC_STATIC, SS_GROUPBOX |
          WS_GROUP | WS_VISIBLE
       CONTROL "Path:", 257, 28, 11, 26, 8, WC_STATIC, SS_TEXT |
          DT_LEFT | DT_TOP | WS_GROUP | WS_VISIBLE
       CONTROL "OK", 258, 185, 31, 38, 12, WC_BUTTON, BS_PUSHBUTTON |
          WS_TABSTOP | WS_VISIBLE | BS_DEFAULT
       CONTROL "*.*", ID_DIRPATH, 57, 11, 166, 8, WC_ENTRYFIELD,
          ES_LEFT | ES_AUTOSCROLL | ES_MARGIN | WS_TABSTOP | WS_VISIBLE
  END
  END


  DLGTEMPLATE IDM_DIREND LOADONCALL MOVEABLE DISCARDABLE
  BEGIN
  DIALOG "", IDM_DIREND, 149, 18, 101, 27, FS_NOBYTEALIGN |
         FS_DLGBORDER | WS_VISIBLE | WS_SAVEBITS
  BEGIN
      CONTROL "Cancel", 256, 30, 2, 38, 12, WC_BUTTON, BS_PUSHBUTTON |
         BS_DEFAULT | WS_TABSTOP | WS_VISIBLE
      CONTROL "Directory Complete", 257, 9, 16, 84, 8, WC_STATIC,
         SS_TEXT | DT_LEFT | DT_TOP | WS_GROUP | WS_VISIBLE
  END
  END
```

End Listing Eighteen

Listing Nineteen

```
HEAPSIZE 16384
STACKSIZE 16384
EXPORTS
    WindowProc
    ChildWindowProc
```

End Listings



The Promise of System 7

Michael Swaine

At MacWorld Expo in Boston this August, System 7 was not to be seen on the exhibit floors. It wasn't shown in the Apple booth, and third-party developers were under an embargo preventing them from showing their System-7-compatible versions or new applications.

This was no great loss, because the major vendors admitted that in their first System 7 releases they will only be concerned with compatibility — making sure their products run under Systems 6 and 7. They will initially implement only those aspects of System 7 that come free: outline fonts, for example. Only later will we see applications that take advantage of significant System 7 features, such as interapplication communication.

Attendees did get a look at System 7, though. At one session, Chris Espinosa, he of the single-digit Apple employee number, and currently the marketing manager for system software at Apple USA, presented what was for most, a first look at System 7. Espinosa and his fellow panelists touched on several points particularly relevant to developers and system administrators; including how Apple is encouraging a fast move to System 7 of its whole user base, and what features of System 7 will support new categories of software. I'll summarize those here, filling in some details that the panelists glossed over.

Espinosa was frank about the Windows threat, acknowledging that very soon there will be as many Windows users as Macintosh users (and implying that soon thereafter there will be a lot more of the former than of the latter). Apple can't afford to fragment its market, Espinosa said; it must deliver a single platform for all Mac software, and a single package of fundamental functionality for all users. System 7 is supposed to become essentially the only operating system version for the Mac. (Although it won't go in the box with all new Macs sold until late next year.) That being the goal, Apple has tried to achieve several objectives with System 7. Apple needs to stay technologically ahead of the competition across its whole line, and needs to provide a platform for a new generation of applications, firing up a lethargic market. And since System 7 is the intended vehicle for reaching both these objectives, Apple needs to handle the introduction of System 7 very well.

Fear of System Software Upgrades

The handling of the introduction is critical, because Apple needs to pick up the momentum it lost when Windows 3 beat System 7 out the door. And if the introduction doesn't encourage people to upgrade right away (rather than waiting for 7.0.1 or 7.0.2, as conventional wisdom dictates), the point at which third-party applications arrive that really take advantage of System 7's new features will be pushed further off.

"The way we have handled system software upgrades in the past has not been all that stellar," Espinosa admitted. But he claims that Apple learned its lesson with System 6.0. "Our fingers were seriously burned," he told one member of the audience. 6.0 was incompatible with a significant number of third-party products and did not offer enough added value to justify the hassle of upgrading. With System 7, Apple has seeded its direct sales sites and application developers early. Developers will have had the product for nine months rather than a few weeks. The sales staff will have the product in beta.

Besides having learned its lesson from the 6.0 release, Apple is in a different position regarding this release than has been the case before. Usually, Apple system software upgrades have been tied to new hardware and couldn't be discussed; this time, Apple can talk more freely, and is doing so.

The 7.0 release is complex enough to need more in the way of documentation, and Apple is aware of this. System 7 will be documented in a completely rewritten system software manual, in a delta guide listing the changes, and in various quick references on specific features. It will go out to system administrators with on-disk documentation, upgrade tips and hints, and compatibility notes directly from the field. Perhaps most importantly, Apple is making every effort to solve compatibility problems and list the unsolved ones so that most software will run with System 7.0 when it's released and so that you'll hear about software that doesn't from Apple before you have to learn it for yourself.

The remote-install capability should ease the installation work of those responsible for more than one Mac. It allows the system administrator to put the installer on a server and let the users install the new system on their machines over the network.

But a smooth introduction alone won't make everyone upgrade. Although the software itself will be distributed through the usual channels and perhaps some new means at a nominal cost, upgrading will be expensive for some users. System 7 requires a Mac Plus, SE, SE/30, Portable, II, IIx, IIcx, IIci, or IIfx, or any future Mac; a minimum of 2-Mbyte memory; and a hard disk drive. A 68030 CPU or a PMMU is necessary for virtual memory, but machines without a 68030 or PMMU will run all other features of System 7. Espinosa says that the goal is to allow the user with 2 Mbyte to print from HyperCard or an application of similar size.

When System 7 is released around the end of this year, the immediate user appeal will be better type, more memory, and more ease of use

Anyone who doesn't have such a system will have to buy hardware to upgrade to System 7, and that's a serious consideration. Even more critically, many of those who meet the minimum requirements will need more RAM to do serious work, and Apple's policy on RAM pricing has a history of being user-hostile. This will have to change, and Apple will have to give users some breaks in upgrading to System 7.

This is tough for Apple; it represents a shift in mindset, and those of us who think Apple needs to get a lot more competitive in pricing should continue to encourage this transition. The relatively generous II-to-fx upgrade price is encouraging, and the new low-cost Macs to be announced this month should be further evidence that Apple is changing.

The other thing needed to make people upgrade immediately is a compelling set of immediate benefits. 7.0 has that.

(continued from page 124)

Building a User Base

Espinosa ran through a demo, emphasizing the new features from a user viewpoint. It's worth examining a few of these immediately-available user features with this question in mind: Does this feature make the user more willing to spend the money and take the perceived risk to upgrade immediately?

The Mac has always been a model of user-interface consistency, and System 7 makes the GUI more consistent and more powerful

Espinosa started by saying that users at Apple often start using another person's Mac without realizing they're working on System 7. At first this sounds like bad news: The product does not distinguish itself. But it's really part of the best user feature of System 7: consistency.

The Mac has always been a model of user interface consistency, and System 7 makes the GUI more consistent at the same time it is becoming more powerful.

The clearest evidence of this is the tightening of the definition of icon behavior. Now, more than ever before, an icon is an icon. When you double-click on it, it does something. As before, applications launch and containers open when double-clicked, but now desk accessories can be launched by double-clicking, too, and the trash acts like any other container, rather than emptying itself whimsically and relocating itself on the desktop when you reboot.

The Apple menu is now associated with a folder of icons, and when you put an icon in the folder, it immediately becomes available in the Apple menu. You can put all sorts of things in the menu this way, including applications and AppleShare volumes.

The System file, which previously gave an error message when double-clicked, now opens to show its contents, including fonts. Double-click on a font and you see an example of the font.

(continued on page 129)

(continued from page 126)

Another improvement in consistency is that MultiFinder is always on under System 7. It will always be possible to access the Finder or other open applications via the icon at the right end of the menubar, which Apple calls a "cooperative multitasking icon." Since it presents a menu of processes, I offer Apple the infinitely more memorable "Process Server." No charge.

The balloon help system is another immediately-accessible advantage of System 7. Click on the help icon in the menubar, and any Finder object you point at displays a cartoon-style word balloon explaining what it is and what it does. You can even pull down menus and inquire about individual menu items. Pointing to a grayed (disabled) menu item will bring up a balloon telling that the item is grayed because it is not presently available, explaining why it is not available, and what you need to do to make it available.

Any Macintosh user who plays with System 7 for a short time is going to walk away with the impression that it is more consistent, more coherent, and more elegant.

Also immediately useful to the user is the new file system, built around the Alias manager. The Finder now finds files. And users can use aliases to catalog disks, which is worth the price of a utility alone and easier than any such utility could possibly be.

Outline fonts will benefit users immediately in one way: they'll get single-point font increments on screen. A smart dealer could make a point by demonstrating this to users, since it distinguishes the Mac interface from Windows. Otherwise, the immediate effect of outline fonts is likely to be negative unless Apple can minimize confusion and incompatibility in the use of different font types. Espinosa described seeing documents with three kinds of fonts (Adobe, bitmapped, TrueType) mixed together. Apple is at least aware of the potential for confusion and incompatibility, and has had a lot of time to work out solutions.

File sharing is another immediate user benefit because it is user-friendly. FileShare works just like the Finder: You just put things in folders. With FileShare, users can pass files around, but they can also pass whole hard disks around: A user can log onto his or her own machine from somebody else's when he or she is in another part of the building, for example.

Espinosa also demonstrated aliasing. "I can leave my system running," he said, "go across the Charles River to our Cambridge office, find a System 7 machine and put my diskette into it, and see an icon of my hard disk back on my desk in Cupertino. I can double-click that hard disk and it'll go out over the network, find the right zone, find my machine, log me in, ask for a password, and open my hard disk." While the actual functionality is remote login with transparent file transfer, it presents itself to the user as: "see an icon, double-click it, and it opens up." The Macintosh way.

So when System 7 is released around the end of this year (if Apple pulls that off), the immediate user appeal will be better type, more memory or applications without more RAM (virtual memory), and more ease of use.

Building System 7 Apps

Ultimately, what will make System 7 pay off is the new applications that will be developed to take advantage of its capabilities. All the nice user features can be viewed as bait to hook as large a user base as possible as quickly as possible for these new applications. As I see it, Apple has provided at least two important categories of support for such applications.

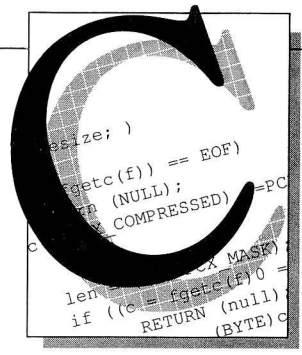
The biggest item is interapplication communications. System 7 has, as I've mentioned here before, several levels of IAC, including live cut-and-paste (Apple calls this "Publish and Subscribe"), in which changes in the source are reflected in the destination; AppleEvents, a set of events that can be passed between applications so that applications can export their functionality to other applications; and low-level inter-process communications, which will allow application developers to build tightly integrated suites of applications. Apple sees these capabilities as laying the groundwork for a new generation of applications: smaller, more focused, and with more opportunity for small add-on products that use the functionality of larger applications.

The other area of support for a new generation of software has to do with multimedia. The new support for sound is impressive, but what's more fundamental is the planned support for time-sensitive information in the form of a set of standards. The umbrella term for this effort is QuickTime.

Apple is soliciting developer input in the definition of both QuickTime and AppleEvents. Since these technologies will help to define the shape of the next generation of applications, it might be wise to take them up on this.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 10.**



Dear Al . . .

Al Stevens

Every now and then I spend time catching up on the letters from readers. In recent times my mailbox has contained some interesting comments on the contents of the column and the C language. Being hard-pressed for new copy this month (July) and as a result of a time-consuming boondoggle trip to a jazz festival in England, I thought I'd cop out and let some of you do my job for me.

Here then is the mostly reader-written "C Programming" column for this month.

Token Pasting

In two previous columns I discussed the ANSI preprocessor ## token-pasting operator, complaining that its purpose is not obvious. Even though the ANSI documentation explains the behavior of ##, it does not adequately explain the operator's reason for being. Several readers responded with their comments, and some sent code that illustrates their use of the ## operator. Because those readers were good enough to enlighten me, I thought it only proper that I should pass the good news on to the rest of you.

Robert L. White of Danbury, Connecticut presents an example from a menu interpreter: *I use [the ## operator] frequently in macro definitions to simplify the generation of tables. It tends to make the differences in tables stand out better.*

Here's an example from a menu interpreter. This thing used tables to describe various menus. In order to keep our sanity, we developed a naming convention. Lists of valid keys for a given menu went in the XXX_Text array, and lists of functions went in the XXX_Funcs array. The addresses of these various lists were stored in a structure called a Menu Descriptor.

```
struct MenuDesc
{
    int     *keys;
    char    **Text;
    int     (**Funcs)( );
};
```

We could have had a table that looked like this:

```
struct MenuDesc AllMenus[] =
{
    Main_Keys, Main_Text, Main_Funcs,
    Brain_Keys, Brain_Text, Brain_Funcs,
    Bone_Keys, Bone_Text, Bone_Funcs,
/*  . . .  etc.  . . .  */
};
```

But, using the following macro, we made the table simpler.

```
#define  MENU(name)\
        name##_Keys, name##_Funcs,
    struct MenuDesc AllMenus[] =
    {
        MENU(Main),
        MENU(Brain),
        MENU(Bone),
    /*  . . .  etc.  . . .  */
    };
```

Our tables were actually much more complex than this, but this gives you an idea of how we used the token-pasting operator to simplify the source code. Token-pasting also increases the code's validity by eliminating opportunities for misspelling.

Mr. White's application of the ## operator hits home because his MenuDesc structure closely resembles the one that I used in the menu drivers for the SMALLCOM project last year. His example illustrates one of the strengths of the ## operator — its ability to reduce repetitious code and therefore reduce the potential for coding errors.

Fred Smith of Stoneham, Massachusetts sends us some history: *I have just re-read your column in the July 1990 issue, and note that you are apparently unaware of the origin of token pasting in C. Don't feel bad, for that is certainly not widespread information. It may be that I don't have all the details either, but here's what I know about it: There is at least one preprocessor in widespread use on Unix systems (I have heard it called "the Reiser preprocessor") which allows the use of an empty comment "/**/" as a token-pasting operator in preprocessor statements. This preprocessor is used on (at least) many BSD-derived systems, such as Sun workstations. Example:*

```
#define pasteup(a,b) (a/**/b)
```

to give you the "Reiser" equivalent of your example on page 134 of the July issue.

This is a horrible kludge, as it directly violates what K&R said about what the preprocessor does with comments, i.e., comments are replaced with white space. Clearly, at least in the case of an empty comment, this preprocessor actually replaces them with nothing. Now, this is occasionally useful, but it certainly isn't C! (to say nothing of being highly non-portable!).

At my last job I had the (mis)fortune to be involved with a (very) large C program which was full of non-portable Sunisms and BSD-isms, including this one. An interesting use was made of the token-pasting hack, though. It was used to create a general-purpose queue (or linked list) management package implemented entirely in the preprocessor.

There were at least a dozen macros defined as part of the package — one

for creating in the preprocessor the declaration of the data structures you wanted, one to initialize it, one to add a new element, one to remove an element, macros for traversing the list, allocating space for each structure, etc.

One of the strengths of the ## operator is its ability to reduce repetitious code and therefore reduce the potential for coding errors

The following is a sample macro which shows the (simplified) flavor of these macros:

```
#define Q_NEXT(a,b)\
                (Q_/**/a ->b.nextchild)
```

so that if this were used in a program like this:

```
child=Q_NEXT(pending_illo_list,
                        current_item);
```

the expansion would look like this:

```
child=(Q_pending_illo_list ->
                current_item.nextchild);
```

I admit that this capability is useful and powerful. However, I think that this kind of power is easily abused because it makes for code that can be extremely terse and difficult to read, even with the macro expansions in hand (note that the actual macros in the package I describe were much more complex than the example I give here). Because of both the obfuscation factor, and the blatant non-portability of token pasting (either the ANSI sort or the Reiser sort) I would think, not twice, but more like twenty times before perpetrating such code.

Mr. Smith goes further to reveal that the pre-ANSI token-pasting technique I found in a C++ header file works with Microsoft C 5.1 and fails with QuickC 2.0. His linked-list example hints at ways to use the preprocessor token-pasting to build a primitive form of parameterized data types, an issue currently being addressed by the C++ community.

Steve Pritchard of Willowdale, Ontario uses the ## token paster to hide the elements of C syntax and make the operative values more prominent when defining a table. He tells us his design philosophies: *In programming I try to use the compiler to assist me in making code understandable and maintainable. I know there are diversities of thought as to whether this means using exotic tricks or just using the basic compiler functions. In the style of coding I use, I sequence in the following priority:*

1. *Removal of multiple, scattered dependencies/references*
2. *Removal of redundant information*
3. *Keeping things simple*

Mr. Pritchard sent a lot of code with his letter. I will try to capture the essence of his token-pasting ideas with this fragment that uses the ## operator to eliminate clutter in a table of hexadecimal values.

```
#define h(n) 0x##n
#define tbl(a,b,c,d) h(a),h(b),h(c),
                                h(d)
int mytable[] =
{
    tbl(02,09,2a,ff),
    tbl(00,1a,1b,e5)
};
```

This code is not an exact extract from Mr. Pritchard's examples, which are in the context of a bigger problem, irrelevant to this discussion. I took the liberty of recasting the technique into a more general example.

There is room for debate about whether the elimination of the *0x* notation in the table initializers adds to readability or confusion. By hiding the C syntax, you could mislead the reader of the code. That *02* initializer looks suspiciously like an octal constant. The *ff* looks like an identifier. Others look like syntax errors. The camouflage is effective only if the reader knows how the coder used the preprocessor. But there is the germ of an idea here. You can use this technique effectively to initialize unusually big tables and thus make the code easier to read but only if you use eye-catching comments and place the macros nearby the tables where the reader can readily find them.

Probably the most ambitious of the token-pasters is Gregory Colvin of Boulder, Colorado who writes: *I have found token-pasting invaluable for automatically creating the many declarations necessary to imitate C++ style objects in C code.*

We are using macros to implement a user interface library that needs to

(continued from page 132)
be portable across all the popular GUI systems. Having spent the last year working in Object Pascal and C++ on the Mac, I didn't want to give up object-oriented programming. But my team at work insists on sticking with C. These macros have proven a reasonable compromise. I also found they had the virtue of removing some of the mystery from object-oriented programming by making the implementation much more apparent.

Programmers and authors should know that thieves have franchised a significant segment of their marketplace — Scurvy dogs

Mr. Colvin sent several source files that do what he requires. The code includes definitions of a base member class complete with common methods. Then it implements a form of class inheritance by defining a CLASS macro that associates the inherited members and methods of a child class to those of the base. The ## operator pastes the common elements of the various structure names to that of the specific class. I have not used Mr. Colvin's approach but have often thought that such a technique might be possible. I have included the three source files, class.h, class.c, and test.c as Listing One, page 147, Listing Two, page 148, and Listing Three, page 148 for your enjoyment and experimentation.

Greg Colvin works at Information Access Systems in Englewood, Colorado. The group he works in uses a version of this code in a standard user interface tool kit.

An Author Responds

I reviewed two MIDI books by Jim Conger in July. He sent me a nice thank-you letter wherein he corrected my impression of his views of computers and people. Here, in part, is Jim's response.

Your quotes about computers and MIDI threatening musicians are actually from the preface written by my able technical editor, Mark Gavin. I

(continued from page 134)
stuck to writing code and explanations.

My own vision of the future role of computers in music is less imaginative than that described in your column. I do not see computers replacing composers any more than I can see computers replacing authors. Like word processors of music, MIDI sequencers have made the task of writing much more efficient and (to me) more satisfying. The authors and composers remain.

MIDI equipment can replace performers, but with mixed results. Even my best efforts on an expensive MIDI setup are far less expressive than those on a 300 year old wooden flute. Synthetic music seems best in the background of a film, not on the front of a stage.

My late father, also a programmer, continually encouraged me to try algorithmic composition. This would seem a natural for someone like myself, long trained in both programming and music. After a few experiments I discarded the sterile outputs, similar to the nonsense poetry written by word pattern analyzers. Intellectually interesting once, but of no emotional content.

The limits of what computers can do seem to result from the limits built into people. We are not beings of logic but are beings of emotion. Music addresses our emotions directly and fails when it attempts to reach us through our conscious thoughts. Millions of years of patterning lead us to a person, talking, singing, or performing with an instrument. Computers will have a difficult time filling this role.

Binding Codes and Messages

The letter from Robert White also included this gem: *I have what I think is an innovative C programming technique that I want to pass on to other programmers. Some of my fellow programmers think it is too exotic because it uses the macro preprocessor and enumerated constants which many C programmers (can you believe it?) think are advanced elements of C language.* [Not *DDJ* readers, Bob. — AS]

The idea is to guarantee that symbolically named indexes into a table always reference the correct element of the table even after several iterations of the code. Sometimes this is hard to guarantee when several developers are working on the same project.

The classic example of such a pairing of index to object is the use of error codes to look up error strings. Here is the simple way of doing this.

```
/* ----- ERRCODE.H ----- */
#define SUCCESS          0
#define BrainError       1
#define BoneError        2

/* ----- ERRTEXT.C ----- */
char *ErrorStrings[] =
{
    "No error",
    "Brain error",
    "Bone error"
    };
```

What if one of the programmers on the project has an old version of the list of error codes and doesn't know it? He or she could spend a good portion of the day trying to track down the wrong error.

Wouldn't it be wonderful if there was a way of specifying the error code and the error string on the same source code line? If the same source code line specified both the code and the string they would always be in sync.

Here's how to do it. We construct a new include file that will serve double duty. When we want it to generate the error strings, we will set a flag for the preprocessor and have the preprocessor spit out a list of error strings. If the flag is not defined, we will force the preprocessor to spit out a set of enumerated constants guaranteed to match the list of error strings.

We do this by creating two definitions for a simple macro. If the flag is

defined, the macro expands to the first argument only. If the flag is not defined, the macro expands to the second argument.

```
#if defined MakeTheStringTable
#define Err(string,code) string
#else
#define Err(string,code) code
#endif
```

The include file will contain the following invocations of this macro:

```
/* ----- ERRCODE.H ----- */
Err("No Error",        SUCCESS),
Err("Brain Error",     BrainError),
Err("Bone Error",      BoneError)
```

To define the table of error messages we now do this:

```
#define MakeTheStringTable
char *ErrorStrings[] =
{
   #include "errcode.h"
);
```

To define the table of indexes we do this:

```
#undef MakeTheStringTable
enum ErrorCode
{
   #include "errcode.h"
};
```

There is no need to limit this technique to generating constants and tables. It is really an aligning technique for any set of lists. The most common way of aligning various lists is to combine them into records or structures. This way you only have one list and all related information is defined on one source line. However, sometimes this is not possible or desirable and lists must be defined as separate arrays. Through a cleverly defined macro inside an include file, elements of both lists can be defined on the same line of source code.

This solution is simple and effective. It solves a problem that I've had maintaining error codes which were implemented exactly the way that Mr. White suggests.

TopSpeed C

I looked at TopSpeed C in April, and JPI responded with a letter to the editor in August. I chided TSC for not allowing you to call an interrupt function the way that Turbo C and Microsoft C do. Without this capability, an interrupt-driven program cannot always chain interrupts. I mentioned this deficiency to the president of JPI in February, and he agreed then that it needed to be addressed. JPI reports in their letter, however, that the *_chain_intr( )* function provides a way for you to call an interrupt function in TSC. Not always enough. If the documentation is correct, this function chains — rather than calls — the called interrupt function to a calling interrupt function. This approach has two deficiencies. First, you can execute an interrupt function only from within another interrupt function. Second, the executed interrupt function returns to the place from where the calling function was invoked, not to the calling function itself. There are many circumstances where an interrupt driver needs to directly call an interrupt function which returns to the calling function. TSC does not allow this, and that restriction renders TSC useless for the development of many interrupt-driven programs.

Avast, Maties

The pirates thrive, and I jump from my mailbox to my soapbox. A letter from Kwee S. Phua of Victoria, Australia tells me he bought two of my C books and needs to get the source code. Dr. Phua says both books were printed in Asia by TP Publications of Singapore. Dr. Phua found my name in *DDJ* and wrote me here. I have never heard of TP Publications and neither has MIS Press, the publisher of my books. But apparently when TP builds illegal copies of computer books, they lack the grace to include the diskette order forms. I guess they are too stupid to bootleg the diskettes as well. Bob Williams, the boss at MIS Press, tells me the Singapore book stores are loaded with pirated copies of books by Norton, Duncan, Schildt, and other notable authors. I shall not feel left out now, being in such prominent company. But Bob also says that nothing can be done about it. Programmers and authors should know that thieves have franchised a significant segment of their marketplace — Scurvy dogs.

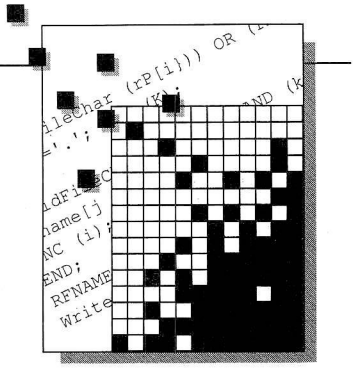
Source Code Availability

As a service to our readers, all source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Drive, Redwood City, CA 94063, or call 800-356-2002 (inside Calif.) or 800-533-4372 (outside Calif.). Specify issue number and disk format. Code is also available through M&T's Telepath online service (via TYMNET) and through the DDJ Forum on CompuServe (type GO DDJ).

DDJ

(Listings begin on page 147.)

Vote for your favorite feature/article.
Circle Reader Service **No. 11.**



Sex and Algorithms

Jeff Duntemann, KI6RA/7

Regardless of obvious empirical evidence as close as any mirror, the fact that our parents had sex often astonishes us. Sex is something *we* invented, after all — nothing so dangerous, messy, or purely delightful could be more than a few years old.

(And the circle is unbroken. All over America, balding and crow-footed exhippies are trying earnestly to explain human reproduction to their nearly-pubescent children, who then can only exclaim, "You mean you and Daddy did *that*?!?! Eeeeee-yukkh!")

I realized recently that I had been guilty of a similar sort of chauvinism when I made use of a day-of-the-week function someone had given me in Pascal/MT+ for CP/M-80 almost ten years back. The algorithm involved was called "Zeller's Congruence," and was as totally opaque to me as the window of a Congressman's limo. Regular readers of this column might remember a note tucked into the comments of my "when stamp" object presented in the April 1990 installment:

"Note that this particular algorithm turns into a pumpkin in 2000. BTW, don't ask me to explain how this crazy thing works. I haven't the foggiest notion. If I ever meet Mr. Zeller, I'll ask him."

I was not being facetious. I pictured Zeller as a distinguished grey-bearded researcher in the CS department at MIT, Yale, or some other exalted institution of higher learning, with a Unix terminal on his desk, and lifetime tenure. I figured I might in fact meet him at a conference or seminar, and I really did intend to ask him how his crazy software gizmo pulled dates out of the air.

Sadly, I won't get the chance. Herr Zeller has been dead for almost a hundred years. On learning that, my first thought was, "Cripes! He didn't even have a computer! How could he possibly have come up with an algorithm?"

You mean people were doing algorithms in 1887? Eeeee-yukkh!

Back to the Source

They were indeed. And just as we used to make numbers in the pre-calculator era by rubbing two sticks together, people managed some amazing mathematical feats in times past with nothing more than pencil, paper, and persistence. Zeller was one, and at least one of his numerical recipes has come to dominate its niche in the software ecosphere.

There are some problems. Several mutant versions of Zeller's Congruence are kicking around. All are utterly opaque, and some of them work better than others. This bothered me, since my own copy ceased to work correctly as of January 1, 2000. Because I didn't know how it worked, I couldn't even begin to try to fix it.

I would have left it at that, but for the help of my friend Hugh Kenner, who did what any good researcher would do and went right to the source. The source was a paper published in German by Zeller in "Acta Mathematica" #7, Stockholm, 1887. Hugh translated the essence of the very brief paper for me, and I implemented it in both Pascal and Modula-2.

The Expression from Hell

I find it a little strange that the current versions of Zeller that I have seen differ so thoroughly from the original recipe. Most versions you see today revolve around a very strange expression ripe with floating-point values:

```
Trunc(Int(2.6 * Month - 0.2))
```

Not only does this seem utterly arbitrary, I don't trust it. I distrust integer-output algorithms that use floating-point constants like this internally, because differences in the way compilers and computers handle floating-point values can easily cause the output to be off by one.

And most remarkably, Zeller used no such floating-point constants in his paper. As simply as I can put it, Zeller laid out the algorithm like this:

Given:
J = Century (e.g. 19)
K = Year (e.g. 90)
q = Day of the month
m = Month, but with a twist: March is still month #3, but January and February are considered months #13 and #14, but of the previous year. In other words, if you're going to calculate a day of the week value for a date in January of 1990, you must call the month #13 and the year 89.

Assuming the Gregorian calendar, we evaluate the expression in Example 1. We then divide the value of the expression by 7 and use the remainder as the index of the day of the week, with 1 = Sunday and 2 = Monday, and so on.

No floating-point constants. Now while it's true that division introduces remainders into the calculation such that it's not truly an integer operation, Zeller himself (in the example he used to illustrate his algorithm) simply took the dividend in each term and ignored the remainder. This is the equivalent of using the DIV operator in Pascal and Modula-2. Look, Ma, no real numbers!

Almost Comprehensible

What I've shown you is how it's done. Zeller says very little in his paper about how it works. In broad terms, the algorithm describes how the day of the week advances. Most people realize that for every year, the day of the week for a given day generally advances by

1; that is, June 29 is on a Friday in 1990, but it will be on a Saturday in 1991. Other things, however, can alter this advance of the day of the week. In 1992, June 29 will be on a Monday rather than the expected Sunday — because 1992 is a leap year, and the extra day in February pushes things ahead by one additional day.

So it's easy to understand the K term, since for every year the day of the week advances by one. Ditto the K/4 term, which tosses in an extra day every four years, and the J/4 term, which tosses in an extra day every four hundred years, when the "century year" leap year that is ordinarily skipped is instead observed. (1700, 1800, and 1900 were not leap years; however, 1600 and 2000 are.) The day of the month term q advances the value from the start of the month.

The peculiar term (m+1)*26/10 advances the count to the start of the given month. The term (which is in fact the single most right-brainedly brilliant part of the whole shebang) compensates for the fact that the months have different numbers of days in them. The twist in the month numbering (that part about making January and February months 13 and 14 instead of 1 and 2) serves the Expression from Hell by putting the most pathological month of all (February) at the end of everything. This allows the oscillations of the leap years to be accounted for elsewhere in the expression, since the (m+1)*26/10 term only takes you to the beginning of a given month. Variations in the length of February thus stay out of the (m+1)*26/10 term.

The one part of the calculation that I truly don't understand is the subtraction of the 2*J term. It's there. The algorithm doesn't work without it. So be it.

$$q + \frac{(m+1)*26}{10} + K + \frac{K}{4} + \frac{J}{4} - 2*J$$

Example 1: *Evaluating the expression for the Gregorian calendar.*

Pumpkin Problems

My own from-the-source implementation of Zeller is contained in the function *CalcDayOfWeek* in Listing One (page 149, which is little more than a frame allowing you to enter different dates to try it out. I encourage you to try to break it, and let me know if it breaks. So far it has worked for every date I've tried (comparing the produced day-of-the-week value against that shown in the Sidekick calendar).

Notice that I have implemented the calculation in a leisurely fashion, accumulating the sum in the intermediate variable *Holder* rather than rolling it all up again into something big and mathematical-looking. My purpose is to make the workings clearer, and because you wouldn't ever need to call this thing from inside a tight loop, the lack of performance shouldn't get in the way. Feel free to pass it through your own code-compactors if you like.

The roach in the Riesling has always been that development for Windows is needlessly difficult

There was one infuriating detail left unmentioned by Zeller in his paper. For certain dates (mostly in March early in a given century) the result of the full expression comes out negative, because the 2*J term is greater than everything else taken together. For every date in which the expression went negative, the day of the week value was off by one. This was actually the reason my inherited implementation of Zeller turns into a pumpkin in the year 2000.

This sounded oddly familiar. I went back to my files and found a nice letter from Carl E. Ohlen of Corvallis, Oregon, pointing this out in the wake of my April column. Carl suggested that when the expression went negative I should add 7 to the expression until it became positive, and only then use the MOD 7 operation on it. I tried it. It works.

Zeller says nothing about such cases, but he was a mathematician and must have been aware of them. If anyone can explain this lapse, I'd like to hear it.

TopSpeed Modula-2 Objects

The Modula-2 implementation of Zeller's Congruence is embedded in Listing Three, page 149, which is my when stamp object implemented in Version 2.0 of TopSpeed Modula-2.

Objects are a natural in Modula-2. They are a natural refinement of the inherent modularity of the language, which has always bundled code and data together at the logistical level by placing them together in modules. A Modula-2 object is rather like a module turned into a data structure, with the addition of inheritance and late binding. At least that's a good place to start thinking of it if you have been on the sidelines watching Turbo Pascal give birth to objects in the structured programming world.

JPI has done an extremely good job implementing objects in Modula-2. What they have done is very close to the Turbo Pascal 5.5 object extensions in many ways. A good place to start is with Listing Two, page 149, which is the definition module for the module implementing the when stamp. The definition of type *When* is nearly identical to the definition of *When* in Turbo Pascal. The main difference is the use of the *CLASS* reserved word rather than the *OBJECT* reserved word. TopSpeed Modula-2 calls object types classes rather than object types, falling into line with Smalltalk, Actor, and C++. While I thought that the "object type" coinage was a good idea at first, I've reversed myself. Object types are classes. We should probably call them classes across the board. Calling an object type a class reduces the inevitable confusion between an object type and an object instance.

Where TopSpeed differs most from Turbo Pascal is in the implementation syntax. Turbo Pascal 5.5 uses a qualifier in the method header to indicate connection with a particular object definition:

```
PROCEDURE When.PutNow;
```

TopSpeed Modula-2 simply enlarges the headers-only class definition given in the definition module to include the bodies of the methods. Everything is bracketed between the two ends of the definition (see Example 2). There is no qualifier on the method header. Because the method is fully implemented between *When = CLASS* and *END*, the compiler always knows to which class a method implementation belongs.

In a reflection of Modula-2's module structure, class definitions minus method bodies are usually placed in definition modules, whereas full class implementations are placed in implementation modules.

Inheritance and Virtual Methods

Most of the other details of TopSpeed Modula-2 objects follow Turbo Pascal nearly to the letter. There is a hidden *SELF* parameter passed implicitly to every method call, providing a connection within a method definition to the

```
TYPE When =
       CLASS
         (* All data field defintions *)
         (* are fully re-stated here. *)

         (* The full method imple-    *)
         (* mentations, including     *)
         (* bodies, are given here.   *)

       END;
```

Example 2: *The two ends of definitions are bracketed.*

particular object instance from which the method call was made.

Inheritance is handled identically to Turbo Pascal, in that the name of the parent class is given in parentheses at the top of the child class definition:

```
TYPE
   KidGadget =
      CLASS(ParentGadget);
```

Late binding is handled through a new reserved word, *VIRTUAL*. Again, just as in Turbo Pascal, a method is made virtual by following the definition of its procedure header with *VIRTUAL*:

```
PROCEDURE Edit( ); VIRTUAL;
```

Calls to virtual methods are handled through a virtual method table, which is a table of code addresses for all virtual methods belonging to an object class. There is one virtual method table for each class definition. Each object instance contains a pointer to the virtual method table belonging to its class. This pointer is not ordinarily visible, and in general it never needs to be referenced. However, a new standard function *MTA( )* returns the address of the virtual method table if you ever need to find it.

In one important departure from the Turbo Pascal scheme, TopSpeed Modula-2 does not implement constructors or destructors.

No Privacy

As with Turbo Pascal, there is no ability to implement a "private" data field or method in the implementation of the class, if that field or method was not originally defined in the definition of the class.

This is unfortunate. Every object should be allowed to have some machinery that it keeps to itself, and that machinery should nonetheless be able to access all other fields and methods belonging to that class. C++ offers classes that ability, and sooner or later such things will have to be added to Pascal and Modula-2 to keep pace.

Nor does TopSpeed Modula-2 implement the multiple inheritance feature that TopSpeed Pascal will contain when released. JPI is hinting that multiple inheritance will be migrated to their Modula-2 in a future release, once their new Pascal compiler proves out the technology.

Listings Two and Three will provide you with a detailed look at the syntactic conventions surrounding object definition and encapsulation. The code is a straight-line translation of the Turbo Pascal when stamp. The only procedure that has been seriously reworked is *CalcDayOfWeek*, which I rewrote from scratch in Modula-2 as well as in Pascal, according to Zeller's original published algorithm.

I haven't had time to really probe the limits of TopSpeed's OOP extensions just yet; in particular, I have some questions involving the creation and deallocation of objects on the heap. You'll get the full report as I figure things out.

In general, I strongly recommend upgrading to TopSpeed Modula-2 V2.0 if you are already a user. Just about all aspects of the product have been greatly improved. The documentation is much improved, with the lone exception of the section on the OOP extensions, which should have been much larger and more detailed. On the other hand, if you've experimented with Turbo Pascal objects, the JPI system will feel pretty familiar and won't take a great deal of additional study.

Jumping through Windows, Take 3

Since last column a pallet load of new software has made its way to Cactus Country, some of it remarkable indeed. Certainly the most important for the long haul is Windows 3.0. I've been a Windows user since well before Windows 1.0 was shipping, back during my tenure at *PC Tech Journal*. Between the product's own gradual improvement and the relentless RAMcharging of our own machines, Windows has become a viable operating platform, far

Products Mentioned

Actor V3.0
The Whitewater Group
1800 Ridge Avenue
Evanston, IL 60201-3621
708-328-3800
Price: $695

ToolBook V1.0
Asymetrix Corp.
110 110th Ave. N.E., Suite 717
Bellevue, WA 98004
206-462-0501
Price: $395

TopSpeed Modula-2 V2.0
Jensen & Partners International
1101 San Antonio Rd., Suite 301
Mountain View, CA 94043
415-967-3200
Price: $199

quicker than any Mac system of comparable price, and better-looking too.

The roach in the Riesling has always been that development for Windows is hideously and needlessly difficult if you're using Microsoft's SDK. I've stated before that Actor is the only reasonable way to develop for Windows, but that's becoming less and less the case.

One promising alternative is Asymetrix's ToolBook, a new development system featuring a brand-new language and a potent suite of resource editing tools. The language reminds me a little of a fully structured and object-oriented Cobol (don't sneer . . . 10 billion lines of code can't all be wrong . . .) and a little bit of the database language Clarion, and that may turn out to be a fine thing indeed.

As with Clarion, you can approach ToolBook development from two directions: From the high end, by using the platform's prototyping tools to build screens and menu structures interactively, while writing as little code as possible, or from the low end, by eschewing the tools and going for the throat with the programming language from the very start.

One thing is for sure: You can build a simple app in ToolBook in almost no time at all, once you've spent an evening getting familiar with the environment. The system makes excellent use of Windows, and (in fact) all copies of Windows 3.0 are being shipped with DayBook, an app written in ToolBook. A look at DayBook will give you an excellent feel for what ToolBook can do.

The downside is that (for now, at least) ToolBook is agonizingly slow. It thrashes the disk constantly for reasons unclear; perhaps loading DLLs, perhaps virtualizing something that I might be able to keep in memory if I had more than my current 4 Mbytes. I don't know just yet, but if you've landed a copy of ToolBook be prepared to grit your teeth a little, and for heaven's sake have a fast 386 with as much RAM as you can afford.

Actor 3.0 showed up a few days ago, with full support for Windows 3.0. A description will have to wait until a future column, but it looks a little faster and has some interesting new features. Sadly, it's significantly more expensive now, and no low-end version is in sight. Still, for my money, Actor remains the way to go for Windows 3.0 development.

Other Windows 3.0 tools have shown up, but most are C-specific and will have to wait on Al Stevens. (Then there's Turbo C++ from Borland — fine piece of work!) Announced but not yet shipping are a Windows 3.0 version of Smalltalk-80 from ParcPlace Systems, and Knowledge Pro Windows, which is difficult to describe and could be the guldurndest thing to turn up this year, judging from the promotional literature Knowledge Pro honcho John Slade sent me recently. More on both when I get my hands on them — but it looks very much like the developer community has decided that for Windows, the third time is the charm.

A Hundred and WHAT?

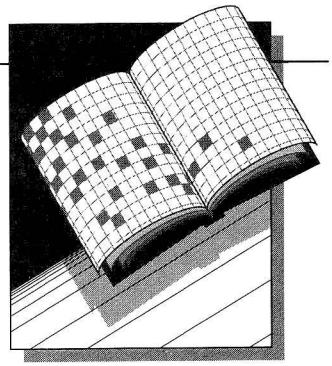
No, you don't want to know. Summer has come to Phoenix. Whereas out in Chicago every June they run pictures in the paper of reporters frying eggs on the sidewalk, out here they don't do that because the eggs catch fire and the neighbors complain about the stink. And as I write it is — I would not kid you about this — 122 degrees.

Now just wait, people say, until July!

DDJ

(Listings begin on page 149.)

Vote for your favorite feature/article.
Circle Reader Service **No. 12.**

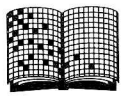


The Theory and Practice of Computer Design and Implementation

Ray Duncan

John L. Hennessy and David A. Patterson's new book, *Computer Architecture: A Quantitative Approach*, is a uniquely lucid and accessible presentation of the theory and practice of computer design and implementation. First, the authors explain basic principles of computer performance measurement and cost/benefit analysis. They then apply these principles to instruction-set design, processor implementation, pipelining, vectorization, memory-hierarchy design, and input and output. Each chapter contains real-life examples drawn from four computer architectures — the IBM 360/370, the DEC VAX, the Intel 8086, and the "DLX" (a sort of hypothetical composite of the currently popular RISC machines) — and each chapter concludes with three entertaining sections entitled "Putting It All Together," "Fallacies and Pitfalls," and "Historical Perspective."

It's almost impossible to convey the authority, scope, and vigor of this book in a brief review. The book is highly technical, of course; nevertheless, it's as engrossing as a novel because of the incredible depth and breadth of the authors' knowledge and experience. It's literally packed with fascinating architectural vignettes, on topics as diverse as the use and misuse of benchmarks, interrupts, microcode, silicon die yields, cache strategies and tradeoffs, and the IBM 3990 I/O subsystem.



Computer Architecture: A Quantitative Approach

John L. Hennessy and
David A. Patterson,
San Mateo, Calif.: Morgan Kaufmann Publishers Inc., 1990
784 pages, hardcover, $54.95
ISBN 1-55860-069-8

In addition, virtually every point in the book is supported and clarified by historical anecdotes which reveal how the important features of modern computer architectures were invented and refined. For example:

> IBM brought microprogramming into the spotlight in 1964 with the IBM 360 family. Before this event, IBM saw itself as many small businesses selling different machines with their own price and performance levels, but also with their own instruction sets. (Recall that little programming was done in high-level languages, so that programs written for one IBM machine would not run on another.) Gene Amdahl, one of the chief architects of the IBM 360, said that managers of each subsidiary agreed to the 360 family of computers only because they were convinced that microcoding made it feasible — if you could take the same hardware and microprogram it with several different instruction sets, they reasoned, then you must also be able to take different hardware and microprogram them to run the same instruction set. To be sure of the viability of microprogramming, the IBM vice president of engineering even visited Wilkes [who built the first microprogrammed CPU in 1958] surreptitiously and had a "theoretical" discussion of the pros and cons of microcode. IBM believed the idea was so important to their plans that they pushed the memory technology inside the company to make microprogramming feasible.
>
> Stewart Tucker of IBM was saddled with the responsibility of porting software from the IBM 7090 to the new IBM 360. Thinking about the possibilities of microcode, he suggested expanding the control store to include simulators, or interpreters, for older machines. Tucker coined the term emulation for this, meaning full simulation at the microprogrammed level. Occasionally, emulation on the 360 was actually faster than the original hardware. Emulation became so popular with customers in the early years of the 360 that it was sometimes hard to tell which instruction set ran more programs.

In spite of such diversions, the book is so well-organized, and the material is developed so logically, that each conclusion as it is reached seems self-evident — even inevitable.

One of the particular strengths of the authors is the deceptive ease with which they transform seemingly simple rules of thumb into razor-sharp analytical tools. For example, they introduce Amdahl's Law within the first few pages: "The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be

used." At first glance, this law appears to be as trivial (and as useless) as the classic syllogism "Socrates is a man, men are mortal, therefore Socrates is mortal." But the authors demonstrate otherwise; they return to Amdahl's Law again and again throughout the book to demonstrate why well-intentioned changes in hardware or software don't always yield the hoped-for benefits. For example:

> Suppose we could improve the speed of the CPU in our machine by a factor of five (without affecting I/O performance) for five times the cost. Also assume that the CPU is used 50% of the time, and the rest of the time the CPU is waiting for I/O. If the CPU is one-third of the total cost of the computer, is increasing the CPU speed by a factor of five a good investment from a cost/performance standpoint?
>
> The speedup obtained is
>
> $$\text{Speedup} = \frac{1}{0.5 + \frac{0.5}{5}} = \frac{1}{0.6} = 1.67$$
>
> The new machine will cost
>
> $$\frac{2}{3} * 1 + \frac{1}{3} * 5 = 2.33 \text{ times the original machine}$$
>
> Since the cost increase is larger than the performance improvement, this change does not improve cost/performance.

At this point, the thoughtful reader might be inclined to reflect on the application of Amdahl's Law to more mundane, practical matters, such as the installation of 80386 accelerator boards into classic IBM PCs with 4.77 MHz, 8-bit I/O buses.

In the last chapter, the authors speculate on future directions in computer architecture, with particular emphasis on multiprocessors and compiler technology. The book ends with a set of appendices that would more than justify the book's price in themselves: A succinct but comprehensive essay on computer arithmetic by David Goldberg of Xerox's Palo Alto Research Center, detailed profiles of instruction set frequencies and execution times for the VAX, IBM 360, and Intel 8086, and a comparative survey of four of the most popular RISC architectures (Intel 860, MIPS, Motorola 88000, and SPARC).

Patterson, a professor at the University of California-Berkeley, was responsible for the design and implementation of RISC I — the direct ancestor of Sun's SPARC processor. Hennessy, a professor at Stanford University, was one of the founders of MIPS Computer Systems and is still that company's chief scientist. The authors are, consequently, legendary figures in the RISC movement, but their discussion of RISC technology in this book is balanced and dispassionate. Much of their profiling and cost/performance data supports RISC concepts, but it's strictly a soft sell — the reader is left in peace to draw his own conclusions. Thus, the book's advocacy of RISC is quite indirect, but is made all the more powerful by the authors' command of every facet of CISC architectures.

One of the particular strengths of the authors is the deceptive ease with which they transform seemingly simple rules of thumb into razor-sharp analytical tools

Computer Architecture: A Quantitative Approach is a tour-de-force on several levels. The book is a masterpiece of technical writing — Hennessy and Patterson's clear, direct style is absorbing and effective, and their enthusiasm for their subject is contagious. The design and production, too, are impeccable. Furthermore, because the book presents a hardheaded and pragmatic approach to computer design, based on real examples, real measurements, and lessons learned from the successes and misadventures of the past, it should revolutionize the teaching of computer architecture and implementation.

Although this book was not written primarily for programmers, it is a thorough and extraordinarily wide-ranging education in that magical interface between the programmer's intentions and the electron's actions. It should be read by every software craftsman who cares about wringing the last drop of performance from his machine.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 13.**

Listing One *(Text begins on page 131.)*

```
/************ CLASS.H COPYRIGHT 1990 GREGORY COLVIN ************
This program may be distributed free with this copyright notice.
*********************************************************************/

#include <stddef.h>
#include <assert.h>

/** All objects must be descendants of the Base class, so we
    define the members and methods of Base here. **/
#define Base_MEMBERS
#define Base_METHODS                                    \
    Base *(*create) (void *table);                      \
    Base *(*clone)  (void *self);                       \
    Base *(*copy)   (void *self, Base *other);          \
    void  (*destroy)(void *self);

typedef struct Base_methods Base_Methods;
typedef struct Base_members Base;
struct Base_members {
    Base_Methods *methods;
};
struct Base_methods {
    char *name;
    size_t size;
    Base_Methods *selfTable;
    Base_Methods *nextTable;
    Base_METHODS
};
extern Base_Methods Base_Table, *Class_List;
Base_Methods *TableFromName(char *name);
Base *ObjectFromName(char *name);

/** The CLASS macro declares a Child class of the Parent. Note
    that Child##_MEMBERS, Child##_METHODS, Parent##_MEMBERS, and
    Parent##_METHODS must be already defined. All methods except
    create() require the first parameter, self, to be a pointer
    to an object of Child type; it can be declared void to stop
    compiler warnings when Parent methods are invoked, at the
    possible cost of warnings when methods are bound. The method
    table, Child##_Table, must be defined as a global structure. **/
#define CLASS(Parent,Child)                             \
typedef struct Child##_methods Child##_Methods;         \
typedef struct Child##_members Child;                   \
struct Child##_members {                                \
    Child##_Methods *methods;                           \
    Parent##_MEMBERS                                    \
    Child##_MEMBERS                                     \
};                                                      \
struct Child##_methods {                                \
    char *name;                                         \
    size_t size;                                        \
    Child##_Methods *selfTable;                         \
    Base_Methods *nextTable;                            \
    Parent##_METHODS                                    \
    Child##_METHODS                                     \
};                                                      \
extern Child##_Methods Child##_Table

/** The INHERIT and BIND macros allow for binding functions to
    object methods at run-time. They must be called before
    objects can be created or used. **/
#define INHERIT(Parent,Child)                           \
    Child##_Table = *(Child##_Methods*)&Parent##_Table;  \
    Child##_Table.name = #Child;                        \
    Child##_Table.size = sizeof(Child);                 \
    Child##_Table.selfTable = &Child##_Table;           \
    Child##_Table.nextTable = Class_List;               \
    Class_List = (Base_Methods *)&Child##_Table

#define BIND(Class,Method,Function)                     \
    Class##_Table.Method = Function

/** The CREATE macro allocates and initializes an object pointer
    by invoking the create() method for the specified Class.
    The DEFINE macro declares an object structure as an
    automatic or external variable; it can take initializers
    after a WITH, and ends with ENDDEF. The NAMED macro creates
    an object pointer from a class name. **/
#define CREATE(Class)                                   \
    (Class *)(*Class##_Table.create)(&Class##_Table)

#define DEFINE(Class,ObjectStruct)                      \
    Class ObjectStruct = { &Class##_Table
#define WITH ,
#define ENDDEF }

#define NAMED(ClassName) ObjectFromName(ClassName)

/** The VALID macro tests the method table self reference. **/
#define VALID(ObjectPtr)                                \
    ((ObjectPtr)->methods->selfTable == (ObjectPtr)->methods)

/** The SEND and CALL macros invoke object methods, through the
    object's methods pointer with SEND, or directly from a
    method table with CALL. Method parameters besides self may
    be sent using WITH, and END terminates the invocation. **/
#define SEND(Message,ObjectPtr)                         \
(   assert(VALID(ObjectPtr)),                           \
    (*((ObjectPtr)->methods->Message))((ObjectPtr)

#define CALL(Class,Method,ObjectPtr)                    \
(   assert(VALID(ObjectPtr)),                           \
    assert(Class##_Table.selfTable == &Class##_Table),  \
    (*(Class##_Table.Method))((ObjectPtr)
```

(continued on page 148)

Listing One *(Listing continued, text begins on page 131.)*

```
#define END ))

/** The DESTROY macro invokes an objects destroy() method **/
#define DESTROY(ObjectPtr) SEND(destroy,(ObjectPtr))END
```

End Listing One

Listing Two

```
/************ CLASS.C COPYRIGHT 1990 GREGORY COLVIN ************
This program may be distributed free with this copyright notice.
***************************************************************/
#include <assert.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include "class.h"

Base *BaseCreate(Base_Methods *table)
{   Base *new = (Base *)calloc(1,table->size);
    assert(new);
    new->methods = table;
    return new;
}

Base *BaseClone(Base *self)
{   Base *new =  (Base *)malloc(self->methods->size);
    assert(new);
    memcpy(new,self,self->methods->size);
    return new;
}

Base *BaseCopy(Base *self, Base *other)
{   memcpy(self,other,self->methods->size);
    return self;
}

void BaseDestroy(Base *self)
{  if (self)
       free(self);
}

Base_Methods Base_Table = {
    "Base",
    sizeof(Base),
    &Base_Table,
    0,
    BaseCreate,
    BaseClone,
    BaseCopy,
    BaseDestroy
};
Base_Methods *Class_List = &Base_Table;

Base_Methods *TableFromName(char *name)
{   Base_Methods *table;
    char *pname, *tname=table->name;
    for (table=Class_List; table; table=table->nextTable)
        for (pname=name; *pname == *tname++; pname++ )
            if (!*pname)
                return table;
    return 0;
}

Base *ObjectFromName(char *name)
{   Base_Methods *table = TableFromName(name);
    if (table)
        return table->create(table);
    return 0;
}
```

End Listing Two

Listing Three

```
/** TEST.C  Add one to argv[1] the hard way. This program serves
no real purpose except invoking most of the CLASS macros. **/
#include <stdio.h>
#include <stdlib.h>
#include "class.h"

/** Define My class members and methods. **/
#define My_MEMBERS int stuff;
#define My_METHODS void (*set)(void*,int); int (*next)(void*);
CLASS(Base,My);

/** Define space for My method table. **/
My_Methods My_Table;

/** Define functions to implement My methods. **/
void MySet(My *self, int new)
{ self->stuff = new;
}

int MyNext(My *self)
{ return ++self->stuff;
}
/** A function to be called in main to initialize My method table. **/
void MyInit( void )
{ INHERIT(Base,My);
  BIND(My,set,MySet);
  BIND(My,next,MyNext);
}

/** Make My class do something. **/
My *test( int i )
{ int j;
#if 1
  My *objectPtr = CREATE(My);   /* One way to make an object */
#else
  DEFINE(My,object)ENDDEF;      /* Another way */
  My *objectPtr= (My *)SEND(clone,&object)END;
#endif
  CALL(My,set,objectPtr) WITH i END;
  if (j = SEND(next,objectPtr)END)
    return objectPtr;
  DESTROY(objectPtr);
  return 0;
}

main(int argc, char **argv)
{ int arg = atoi(argv[1]);
  My *out;
  MyInit();
  if (out = test(arg)) {
    printf("%d\n",out->stuff);
    DESTROY(out);
  } else
    printf("0\n");
}
```

End Listings

Listing One *(Text begins on page 139.)*

```
PROGRAM ZelTest;  { From DDJ 10/90 }

CONST
  DayStrings : ARRAY[0..6] OF STRING =
  ('Sunday','Monday','Tuesday','Wednesday','Thursday','Friday','Saturday');

VAR
  Month, Day, Year : Integer;


FUNCTION CalcDayOfWeek(Year,Month,Day : Integer) : Integer;

VAR
  Century,Holder : Integer;

BEGIN
  { First test for error conditions on input values: }
  IF (Year < 0)  OR
     (Month < 1) OR (Month > 12) OR
     (Day < 1)   OR (Day > 31) THEN
    CalcDayOfWeek := -1  { Return -1 to indicate an error }
  ELSE
    { Do the Zeller's Congruence calculation as Zeller himself }
    { described it in "Acta Mathematica" #7, Stockhold, 1887.  }
    BEGIN
      { First we separate out the year and the century figures: }
      Century := Year DIV 100;
      Year    := Year MOD 100;
      { Next we adjust the month such that March remains month #3, }
      {  but that January and February are months #13 and #14,     }
      {  *but of the previous year*: }
      IF Month < 3 THEN
        BEGIN
          Inc(Month,12);
          IF Year > 0 THEN Dec(Year,1)      { The year before 2000 is }
            ELSE                            { 1999, not 20-1...       }
              BEGIN
                Year := 99;
                Dec(Century);
              END
        END;

      { Here's Zeller's seminal black magic: }
      Holder := Day;                        { Start with the day of month }
      Holder := Holder + (((Month+1) * 26) DIV 10); { Calc the increment }
      Holder := Holder + Year;              { Add in the year }
      Holder := Holder + (Year DIV 4);      { Correct for leap years  }
      Holder := Holder + (Century DIV 4);   { Correct for century years }
      Holder := Holder - Century - Century; { DON'T KNOW WHY HE DID THIS! }
      WHILE Holder < 0 DO                   { Get negative values up into }
        Inc(Holder,7);                      { positive territory before   }
                                            { taking the MOD...           }
      Holder := Holder MOD 7;               { Divide by 7 but keep the  }
                                            { remainder rather than the }
                                            { quotient }

      { Here we "wrap" Saturday around to be the last day: }
      IF Holder  = 0 THEN Holder := 7;

      { Zeller kept the Sunday = 1 origin; computer weenies prefer to }
      { start everything with 0, so here's a 20th century kludge:     }
      Dec(Holder);

      CalcDayOfWeek := Holder;  { Return the end product! }
    END;
END;

BEGIN
  Write('Month (1-12): '); Readln(Month);
  Write('Day   (1-31): '); Readln(Day);
  Write('Year        : '); Readln(Year);
  Writeln('The day of the week is ',
          DayStrings[CalcDayOfWeek(Year,Month,Day)]);
  Readln;
END.
```

End Listing One

Listing Two

```
(*-------------------------------------------------------*)
(*                      TIMEDATE                         *)
(*                                                       *)
(* A Time-and-date stamp object for TopSpeed Modula-2    *)
(*                                                       *)
(*                           Definition module           *)
(*                           TopSpeed Modula-2 V2.0      *)
(*                           by Jeff Duntemann           *)
(*                           Last update 6/1/90          *)
(*                                                       *)
(*-------------------------------------------------------*)

DEFINITION MODULE TimeDate;

TYPE
  String9  = ARRAY[0..9]  OF CHAR;
  String20 = ARRAY[0..20] OF CHAR;
  String50 = ARRAY[0..50] OF CHAR;

  WhenUnion =
    RECORD
      CASE  : BOOLEAN OF
        TRUE  : FullStamp : LONGCARD; |
        FALSE : TimePart  : CARDINAL;
                DatePart  : CARDINAL
      END;
    END;

  When =
    CLASS
      WhenStamp      : WhenUnion;       (* Combined time/date stamp *)
      TimeString     : String9;         (* i.e., "12:45a"           *)
      Hours,Minutes,Seconds : CARDINAL; (* Seconds is always even!  *)
      DateString     : String20;        (* i.e., "06/29/89"         *)
      LongDateString : String50;        (* i.e., "Thursday, June 29, 1989" *)
      Year,Month,Day : CARDINAL;
      DayOfWeek      : INTEGER;         (* 0=Sunday, 1=Monday, etc. *)
      PROCEDURE GetTimeStamp() : CARDINAL; (* Returns DOS-format time stamp *)
      PROCEDURE GetDateStamp() : CARDINAL; (* Returns DOS-format date dtamp *)
      PROCEDURE PutNow;
      PROCEDURE PutWhenStamp(NewWhen  : LONGCARD);
      PROCEDURE PutTimeStamp(NewStamp : CARDINAL);
      PROCEDURE PutDateStamp(NewStamp : CARDINAL);
      PROCEDURE PutNewDate(NewYear,NewMonth,NewDay : CARDINAL);
      PROCEDURE PutNewTime(NewHours,NewMinutes,NewSeconds : CARDINAL);
    END;

END TimeDate.
```

End Listing Two

Listing Three

```
(*-------------------------------------------------------*)
(*                      TIMEDATE                         *)
(*                                                       *)
(* A Time-and-date stamp object for TopSpeed Modula-2    *)
(*                                                       *)
(*                           Implementation module       *)
(*                           TopSpeed Modula-2 V2.0      *)
(*                           by Jeff Duntemann           *)
(*                           Last update 6/16/90         *)
(*                                                       *)
(*-------------------------------------------------------*)

IMPLEMENTATION MODULE TimeDate;

FROM FIO     IMPORT GetCurrentDate;
FROM Str     IMPORT CardToStr,Concat,IntToStr,Length,Slice;
FROM Bitwise IMPORT And,Or;  (* From DDJ for March 1990 *)

TYPE
  TMonthTags = ARRAY [1..12] OF String9;
  TDayTags   = ARRAY [0..6]  OF String9;


VAR
  Temp1 : String50;
  Dummy : CARDINAL;
  DayTags : TDayTags;
  MonthTags : TMonthTags;


PROCEDURE CalcTimeStamp(Hours,Minutes,Seconds : CARDINAL) : CARDINAL;

BEGIN
  RETURN Or(Or((Hours << 11),(Minutes << 5)),(Seconds >> 1));
END CalcTimeStamp;


PROCEDURE CalcDateStamp(Year,Month,Day : CARDINAL) : CARDINAL;

BEGIN
  RETURN Or(Or(((Year - 1980) << 9),(Month << 5)),Day);
END CalcDateStamp;


PROCEDURE CalcTimeString(VAR TimeString : String9;
                         Hours,Minutes,Seconds : CARDINAL);

VAR
  Temp1,Temp2 : String9;
  AMPM        : CHAR;
  I           : INTEGER;
  OK          : BOOLEAN;

BEGIN
  I := Hours;
  IF Hours = 0 THEN I := 12; END;   (* "0" hours = 12am *)
  IF Hours > 12 THEN I := Hours - 12; END;
  IF Hours > 11 THEN AMPM := 'p' ELSE AMPM := 'a'; END;
  IntToStr(LONGINT(I),Temp1,10,OK);
  IntToStr(LONGINT(Minutes),Temp2,10,OK);
  IF Length(Temp2) < 2 THEN Concat(Temp2,'0', Temp2); END;
  Concat(TimeString,Temp1,':');
  Concat(TimeString,TimeString,Temp2);
  Concat(TimeString,TimeString,AMPM);
END CalcTimeString;


PROCEDURE CalcDateString(VAR DateString : String20;
                         Year,Month,Day : CARDINAL);

VAR
```

(continued on page 150)

(Listing continued, text begins on page 139.)

```
  OK : BOOLEAN;

BEGIN
  CardToStr(LONGCARD(Month),DateString,10,OK);
  CardToStr(LONGCARD(Day),Temp1,10,OK);
  Concat(DateString,DateString,'/');
  Concat(DateString,DateString,Temp1);
  CardToStr(LONGCARD(Year),Temp1,10,OK);
  Concat(DateString,DateString,'/');
  Slice(Temp1,Temp1,3,2);
  Concat(DateString,DateString,Temp1);
END CalcDateString;


PROCEDURE CalcLongDateString(VAR LongDateString : String50;
                             Year,Month,Date,DayOfWeek : CARDINAL);
VAR
  Temp1 : String9;
  OK    : BOOLEAN;

BEGIN
  Concat(LongDateString,DayTags[DayOfWeek],', ');
  CardToStr(LONGCARD(Date),Temp1,10,OK);
  Concat(LongDateString,LongDateString,MonthTags[Month]);
  Concat(LongDateString,LongDateString,' ');
  Concat(LongDateString,LongDateString,Temp1);
  Concat(LongDateString,LongDateString,', ');
  CardToStr(LONGCARD(Year),Temp1,10,OK);
  Concat(LongDateString,LongDateString,Temp1);
END CalcLongDateString;


(*-------------------------------------------------------------------------*)
(* This calculates a day of the week figure, where 0=Sunday, 1=Monday, *)
(* and so on, given the year, month, and day.  The year must be passed *)
(* in full; that is, "1990" not just "90".  Another century is at hand,*)
(* gang...                                                             *)
(*-------------------------------------------------------------------------*)

PROCEDURE CalcDayOfWeek(Year,Month,Day : INTEGER) : INTEGER;

VAR
  Century,Holder : INTEGER;

BEGIN
  (* First test for error conditions on input values: *)
  IF (Year < 0)  OR
     (Month < 1) OR (Month > 12) OR
     (Day < 1)   OR (Day > 31)
  THEN
    RETURN -1  (* Return -1 to indicate an error *)
  ELSE
    (* First we separate out the year and century figures: *)
    Century := Year DIV 100;
    Year    := Year MOD 100;
    (* Next we adjust the month such that March remains #3,   *)
    (*  but that January and February are months #13 and #14, *)
    (*  *but of the previous year.*                           *)
    IF Month < 3 THEN
      INC(Month,12);
      IF Year > 0 THEN DEC(Year,1)    (* 1900/2000 etc. ("year 0")    *)
        ELSE                          (* must be treated specially.   *)
          Year := 99;                 (* You can't just decrement the *)
          DEC(Century)                (* year to -1...you must make   *)
        END;                          (* it year 99 of the previous   *)
    END;                              (* century.                     *)

    (* Here's Zeller's seminal black magic: *)
    Holder := Day;                                (* Start with the day *)
    Holder := Holder + (((Month+1) * 26) DIV 10);  (* Calc increment *)
    Holder := Holder + Year;                      (* Add in the year    *)
    Holder := Holder + (Year DIV 4);       (* Correct for leap years *)
    Holder := Holder + (Century DIV 4); (* Correct for century years *)
    Holder := Holder - Century - Century;  (* Take out century twice *)
    WHILE Holder < 0 DO     (* Avoid taking MOD of negative quantity *)
      INC(Holder,7);
    END;

    Holder := Holder MOD 7;    (* Take Modulo 7 of (positive) result *)

    (* Here we "wrap" Saturday around to be the last day: *)
    IF Holder = 0 THEN Holder := 7 END;

    (* Zeller kept the Sunday = 1 origin; computer weenies prefer to *)
    (* start everything with 0, so here's a 20th century kludge:     *)
    DEC(Holder);

    (* We've got it: Sunday = 0, Monday = 1, etc. Return the value:  *)
    RETURN Holder;
  END;  (* IF *)
END CalcDayOfWeek;


TYPE
  When =
    CLASS
      WhenStamp      : WhenUnion;       (* Combined time/date stamp *)
      TimeString     : String9;         (* i.e., "12:45a"           *)
      Hours,Minutes,Seconds : CARDINAL; (* Seconds is always even!  *)
      DateString     : String20;        (* i.e., "06/29/89"         *)
      LongDateString : String50;        (* i.e., "Thursday, June 29, 1989" *)
      Year,Month,Day : CARDINAL;
      DayOfWeek      : INTEGER;         (* 0=Sunday, 1=Monday, etc. *)

      (*-------------------------------------------------------------------*)
      (* There will be many times when an individual date or time stamp will *)
      (* be much more useful than a combined time/date stamp.  These simple  *)
      (* functions return the appropriate half of the combined long integer  *)
      (* time/date stamp without incurring any calculation overhead.  It's   *)
      (* done with a simple value typecast:                                  *)
      (*-------------------------------------------------------------------*)

      PROCEDURE GetTimeStamp() : CARDINAL;

      BEGIN
        RETURN WhenStamp.TimePart;
      END GetTimeStamp;


      PROCEDURE GetDateStamp() : CARDINAL;

      BEGIN
        RETURN WhenStamp.DatePart;
      END GetDateStamp;


      (*-------------------------------------------------------------------*)
      (* To fill a When record with the current time and date as maintained  *)
      (* by the system clock, execute this method:                           *)
      (*-------------------------------------------------------------------*)

      PROCEDURE PutNow;

      BEGIN
        (* Get current time and date from the system: *)
        WhenStamp.FullStamp := GetCurrentDate();
        (* Calculate a new time stamp and update object fields: *)
        PutTimeStamp(WhenStamp.TimePart);
        (* Calculate a new date stamp and update object fields: *)
        PutDateStamp(WhenStamp.DatePart);
      END PutNow;


      (*-------------------------------------------------------------------*)
      (* This method allows us to apply a whole long integer time/date stamp *)
      (* to the When object.  The object divides the stamp into time and     *)
      (* date portions and recalculates all other fields in the object.      *)
      (*-------------------------------------------------------------------*)

      PROCEDURE PutWhenStamp(NewWhen  : LONGCARD);

      BEGIN
        WhenStamp.FullStamp := NewWhen;
        (* We've actually updated the stamp proper, but we use the two *)
        (* "put" routines for time and date to generate the individual *)
        (* field and string representation forms of the time and date. *)
        (* I know that the "put" routines also update the long integer *)
        (* stamp, but while unnecessary it does no harm.               *)
        PutTimeStamp(WhenUnion(WhenStamp).TimePart);
        PutDateStamp(WhenUnion(WhenStamp).DatePart);
      END PutWhenStamp;
```

```
(*------------------------------------------------------------------------*)
(* We can choose to update only the time stamp, and the object will      *)
(* recalculate only its time-related fields.                              *)
(*------------------------------------------------------------------------*)

PROCEDURE PutTimeStamp(NewStamp : CARDINAL);

BEGIN
  WhenUnion(WhenStamp).TimePart := NewStamp;
  (* The time stamp is actually a bitfield, and all this shifting left *)
  (* and right is just extracting the individual fields from the stamp:*)
  Hours := NewStamp >> 11;

  Minutes := And((NewStamp >> 5),3FH);
  Seconds := And((NewStamp << 1),1FH);
  (* Derive a string version of the time: *)
  CalcTimeString(TimeString,Hours,Minutes,Seconds);
END PutTimeStamp;


(*------------------------------------------------------------------------*)
(* Or, we can choose to update only the date stamp, and the object       *)
(* will then recalculate only its date-related fields.                    *)
(*------------------------------------------------------------------------*)

PROCEDURE PutDateStamp(NewStamp : CARDINAL);

BEGIN
  WhenUnion(WhenStamp).DatePart := NewStamp;
  (* Again, the date stamp is a bit field and we shift the values out  *)
  (* of it: *)
  Year  := (NewStamp >> 9) + 1980;
  Month := And((NewStamp >> 5),0FH);
  Day   := And(NewStamp,1FH);
  (* Calculate the day of the week value using Zeller's Congruence:    *)
  DayOfWeek := CalcDayOfWeek(Year,Month,Day);
  (* Calculate the short string version of the date; as in "06/29/89": *)
  CalcDateString(DateString,Year,Month,Day);
  (* Calculate a long version, as in "Thursday, June 29, 1989": *)
  CalcLongDateString(LongDateString,Year,Month,Day,DayOfWeek);
END PutDateStamp;


PROCEDURE PutNewDate(NewYear,NewMonth,NewDay : CARDINAL);

BEGIN
  (* The "boss" field is the date stamp.  Everything else is figured *)
  (* from the stamp, so first generate a new date stamp, and then    *)
  (* (odd as it may seem) regenerate everything else, *including*    *)
  (* the Year, Month, and Day fields: *)
  PutDateStamp(CalcDateStamp(NewYear,NewMonth,NewDay));
  (* Calculate the short string version of the date; as in "06/29/89": *)
  CalcDateString(DateString,Year,Month,Day);
    (* Calculate a long version, as in "Thursday, June 29, 1989": *)
    CalcLongDateString(LongDateString,Year,Month,Day,DayOfWeek);
  END PutNewDate;


  PROCEDURE PutNewTime(NewHours,NewMinutes,NewSeconds : CARDINAL);

  BEGIN
    (* The "boss" field is the time stamp.  Everything else is figured *)
    (* from the stamp, so first generate a new time stamp, and then    *)
    (* (odd as it may seem) regenerate everything else, *including*    *)
    (* the Hours, Minutes, and Seconds fields: *)
    PutTimeStamp(CalcTimeStamp(NewHours,NewMinutes,NewSeconds));
    (* Derive the string version of the time: *)
    CalcTimeString(TimeString,Hours,Minutes,Seconds);
  END PutNewTime;

END;  (* ...of CLASS When implementation *)


BEGIN   (* Initialization code for TimeDate goes here: *)
  MonthTags :=
    TMonthTags('January','February','March','April','May','June','July',
               'August','September','October','November','December');
  DayTags := TDayTags('Sunday','Monday','Tuesday','Wednesday',
                      'Thursday','Friday','Saturday');
END TimeDate.
```

End Listings

Implementing Cordic Algorithms

A single compact routine for computing transcendental functions

Pitts Jarvis

Efficiently computing sines, cosines, and other transcendental functions is a process about which many programmers are blissfully ignorant. When these values are called for in a graphics or CAD program, we usually rely on a call to the compiler's run-time library. The library either derives the necessary values in some mysterious manner or calls the floating-point coprocessor to assist in the task.

The CORDIC (COordinate, Rotation DIgital Computer) family of algorithms is an elegant, efficient, and compact way to compute sines, cosines, exponentials, logarithms, and associated transcendental functions using one core routine. These truly remarkable algorithms compute these functions with n bits of accuracy in n iterations — where each iteration requires only a small number of shifts and additions. Furthermore, these routines use only fixed-point arithmetic. Using these algorithms, you can cast your entire graphics application into fixed-point, and thus avoid the cost of run-time conversion from fixed- to floating-point representation and back.

Even if you don't plan on recasting your application into fixed-point, you just might be curious how your floating-point coprocessor works. The Intel numerics family (8087, 80287, and 80387) all use Cordic algorithms, in a form slightly different than described here, to compute circular functions. The Intel implementations are described by R. Nave[1] and A. K. Yuen[2].

The implementations may be contemporary, but the algorithms are not new. J. E. Volder[3] coined the name in 1959. He applied these algorithms to build a special-purpose digital computer for real-time airborne navigation. D. S. Cochran[4] identifies their use in the HP-35 calculator in 1972 to calculate the transcendental functions.

Pitts Jarvis is a senior engineer at 3Com Corporation. He can be reached at 1275 Martin Ave., Palo Alto, CA 94301.

Mathematical Manipulation

If we have a vector $[x, y]$, we can rotate it through an angle a by multiplying it by the matrix Ra, defined in Example 1(a). Explicitly doing the matrix multiplication yields the equation in Example 1(b).

If we choose $x = 1$ and $y = 0$ and multiply that vector by Ra we are left with the vector $[\cos a, \sin a]$.

Multiplying by two successive rotation matrices, Ra and Rb rotates the vector through the angle $a + b$, or more formally $RaRb = Ra+b$. If we choose to represent the angle a as a sum of angles a_i for $i = 0$ through n (see Example 1(c)), then we can rotate the vector through the angle a by multiplying a series of rotation matrices Ra_0, Ra_1, . . . Ra_n.

By picking the a_i carefully, we can simplify the arithmetic. Notice that we can rewrite the rotation matrix by factoring out cos a as shown in Example 1(d). If we pick a_i such that $\tan a_i = 2^{-i}$ for $i = 0$ through n, all of the multiplications by tan a_i become right shifts by i bits.

Now we need to specify an algorithm so that we can represent a as the sum of the a_i. Initialize a variable, z, to a. This z will be a residue quantity, which we are trying to drive to zero by adding or subtracting a_i at the i-th step. At the first step, $i = 0$. At the i-th step, if $z \geq 0$ then subtract a_i from z. Otherwise add a_i to z. At the last step $i = n$, and z is the error in our representation of a. Notice that in Example 1(e), for large i, each additional step yields one more bit of accuracy in our representation of a.

Figure 1 shows the relative magnitudes of the incremental angles, a_i. Figure 2 gives an example of the convergence process with an initial angle of 0.65. Notice that successive iterations do not necessarily reduce the absolute error in the representation of the angle. Also notice that the error does not oscillate about zero.

At each step as we decompose a into the sum or difference of the a_i's we could also multiply our vector $[x, y]$ by the appropriate Ra_i or $R\text{-}a_i$ depending on whether we add or subtract a_i. Remember, these multiplications are nothing more than shifts. We must also multiply in the still embarrassing factor cos a_i. However, cosine is an even

function and has the property that cos a_i = cos ($-a_i$). It does not matter whether we add or subtract the angle — we always multiply by the same factor! Because all of the cos a_i can be factored out and grouped together, we can treat their product as a constant and compute it only once, along with all the $a_i = \tan^{-1} 2^{-i}$.

Not all angles can be represented as the sum of a_i. There is a domain of convergence outside of which we cannot reduce the angle to within a_{n-1} of zero. See Example 1(f). For the algorithm to work, we must start with a such that $|a| \leq a_{max} \approx 1.74$. This conveniently falls just outside the first quadrant. If we are given an angle outside the first quadrant, we can scale it by dividing by $\pi/2$ obtaining a quotient Q and a remainder D where $|D| < \pi/2 < a_{max}$. Since the algorithm computes both sine and cosine of D, we pick the appropriate value and sign depending on the value of Q.

What about angles within the domain of convergence? It's not obvious that the strange set we've picked (see Example 1(g)) can represent all angles within the domain of convergence to within a_{n-1}. But, using mathematical induction, Walther[5] proves that the scheme works.

The Circular Functions

One variation of the Cordic algorithm computes the circular functions — sin, cos, tan, and so on. This algorithm is shown in pseudocode in Example 2(a).

First, start with $[x,y,z]$. The x and y are as before. z is the quantity that we drive to zero, with an initial value of angle a. The first step in the loop decides whether to add or subtract a_i from the residue z. The variable s is correspondingly positive or negative. The second step reduces the magnitude of z and effects the multiplications by the tan a_i. The expression $y>>i$ means shift y right by i bits.

When you start the algorithm with $[x,y,z]$ and then drive z to zero as specified by Example 2(a), we are left with the quantities in Example 3(a). Where K is a constant, it is just the product of the cos a_i, as in Example 3(b).

For the curious, $K \approx 0.607$. The value of K can be precomputed by setting $[x,y,z]$ to [1, 0, 0] and running the algorithm as before. The result is shown in Example 3(c). Take the reciprocal of the final x and we have K. Therefore, to compute sin a and cos a, set $[x,y,z]$ to $[K, 0, a]$ and run the algorithm. Example 3(d) shows the result. In effect, we start with a vector $[x, y]$, and rotate it through a given angle a, by driving z to zero. Running the algorithm with the special case where the vector initially lies along the x axis and is of length K, rotates the vector by angle a and leaves behind cos a and sin a. This relationship is shown in Figure 3.

To compute $\tan^{-1} a$, instead of z, we could choose to drive y to zero. Driv-

(a) $$R_a = \begin{bmatrix} \cos a & -\sin a \\ \sin a & \cos a \end{bmatrix}$$

(b) $$R_a \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \cos a - y \sin a \\ x \sin a + y \cos a \end{bmatrix}$$

(c) $$a = a_0 + a_1 + \ldots + a_n$$

(d) $$R_a = \cos a \begin{bmatrix} 1 & -\tan a \\ \tan a & 1 \end{bmatrix}$$

(e) $$a_i = \tan^{-1} 2^{-i} \approx 2^{-i}, \text{ for large } i$$

(f) $$a_{max} = \tan^{-1} 2^{-0} + \tan^{-1} 2^{-1} + \ldots + \tan^{-1} 2^{-n}$$

(g) $$\pm \tan^{-1} 2^{-0} \pm \tan^{-1} 2^{-1} \pm \ldots \pm \tan^{-1} 2^{-n}$$

Example 1: *Mathematical expressions*

```
(a)  for i from 0 to n do
     {
          if (z ≥ 0) then s:= 1 else s:= -1;
          [x,y,z] := [x - s y>>i,  y + s x>>i, z - s a_i]
     }.
(b)  for i from 0 to n do
     {
          if (y ≥ 0) then s:= 1 else s:= -1;
          [x,y,z]:=[x + s y>>i, y - s x >>i,z +  s a_i]
     }
```

Example 2: *(a) The basic algorithm; (b) the inverse algorithm*

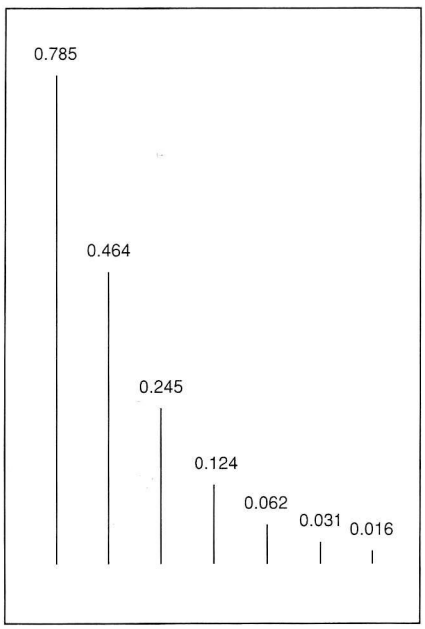


Figure 1: *$\tan^{-1} 2^{-i}$ for i=0,1,2,3,4,5,6*

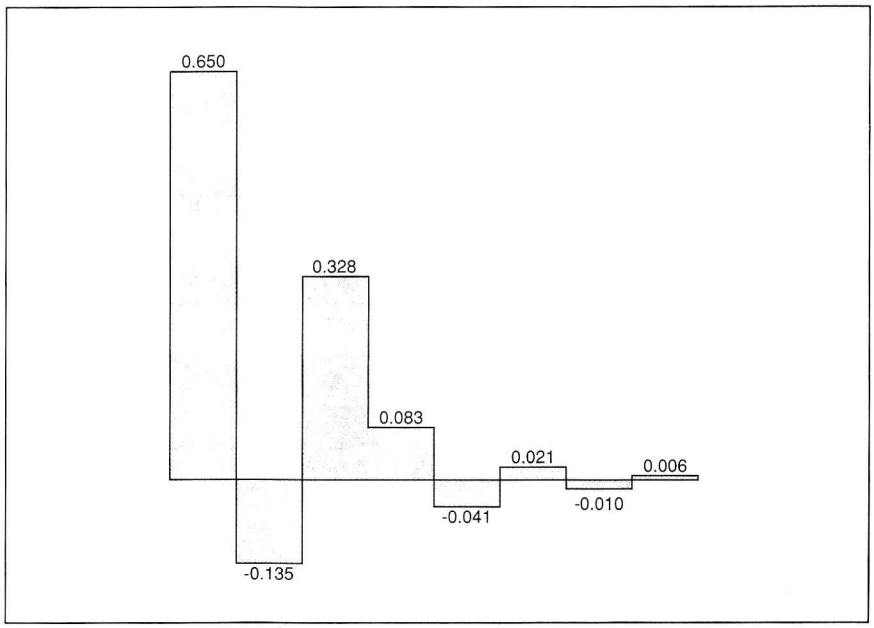


Figure 2: *Example of convergence with an initial angle of 0.65*

ing y to zero rotates the vector through the angle a, the angle subtended by the vector and the x axis, leaving the vector lying along the x axis. Start with the vector anywhere in the first or fourth quadrant and an initial value of zero in z. The first or fourth quadrant is used because almost all vectors in the second or third quadrant will not converge. At the i-th step, if $y \geq 0$, the vector lies in the first quadrant, subtract a_i from z. Move the vector closer to the x axis; rotate it by negative a_i by multiplying by the rotation matrix $R\text{-}a_i$. If $y < 0$, the vector lies in the fourth quadrant, add a_i to z and multiply the vector by Ra_i. At the end, z has the negative of the angle of the original vector $[x, y]$, $\tan^{-1} y/x = \tan^{-1} a$.

Interesting special cases include exponential, square root, and logarithm

Changing the sign of a_i has no effect on the computed values of x and y and leaves the original angle a in z rather than its negative. With this change, the inverse algorithm to drive y to zero becomes the expression shown as in the algorithm in Example 2(b).

Starting with $[x,y,z]$ and then driving y to zero using the inverse algorithm leaves behind the quantities in Example 3(e).

Hyperbolic Functions

The hyperbolic functions (sinh, cosh, and so on) are similar to the circular functions. The correspondences between these two types of functions are shown in Table 1.

By analogy, use H_a as the rotation matrix and represent a using the set $a_i = \tanh^{-1} 2^{-i}$ for $i = 1$ to n. Notice that for hyperbolics, a_0 is infinity.

Given the change in the a_i, can we still represent any angle a within the domain of convergence the same way we did for the circular functions? Unfortunately, the answer is no! Walther points out that repeating an occasional term makes the representation converge in the hyperbolic case. Repeating the terms as shown in Example 4(a) does the trick.

Except for the repeated terms and some changes of sign, the algorithms for hyperbolic functions are identical to the circular functions. Listing One,

(a) $[x,y,z] \rightarrow [\frac{1}{K}(x\cos z - y\sin z), \frac{1}{K}(y\cos z + x\sin z)$g 0]

(b) $K = \cos a_0 \cdot \cos a_1 \cdot \ldots \cdot \cos a_n$

(c) $[1,0,0] \rightarrow [\frac{1}{K}, 0, 0]$

(d) $[K,0,a] \rightarrow [\cos a, \sin a, 0]$

(e) $[x,y,z] \rightarrow [\frac{1}{K}\sqrt{x^2 + y^2}, 0, z + \tan^{-1}\frac{y}{x}]$

Example 3: *Circular function quantities*

(a) $a_4, a_{13}, a_{40}, a_{121}, \ldots, a_k, a_{3k+1}, \ldots$

(b) $[x,y,z] \rightarrow [\frac{1}{K}(x\cosh z + y\sinh z), \frac{1}{K}(\cosh z + x\sinh z), 0]$

(c) $[x,y,z] \rightarrow [\frac{1}{K}\sqrt{x^2 - y^2}, 0, z + \tanh^{-1}\frac{y}{x}]$

(d) $[K,K,a] \rightarrow [e^a, e^a, 0]$

(e) $[a + \frac{K^2}{4}, a - \frac{K^2}{4}, \frac{1}{2}\ln\frac{K^2}{4}] \rightarrow [\sqrt{a}, 0, \frac{1}{2}\ln a]$

Example 4: *Hyperbolic function quantities*

(a) $\{\tan^{-1} 1, \tan^{-1}\frac{1}{2}, \tan^{-1}\frac{1}{4}, \ldots\}$

(b) $\{\tanh^{-1}\frac{1}{2}, \tanh^{-1}\frac{1}{4}, \ldots\}$

(c) $\tan^{-1} x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \ldots$ for $x^2 \leq 1$

(d) $\tanh^{-1} x = x + \frac{x^3}{3} + \frac{x^5}{5} + \frac{x^7}{7} + \frac{x^9}{9} + \ldots$ for $x^2 < 1$

(e) $a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1 x + a_0$

(f) $(\ldots((a_n x + a_{n-1})x + a_{n-2})x + \ldots + a_1)x + a_0$

Example 5: *Quantities for calculating constants*

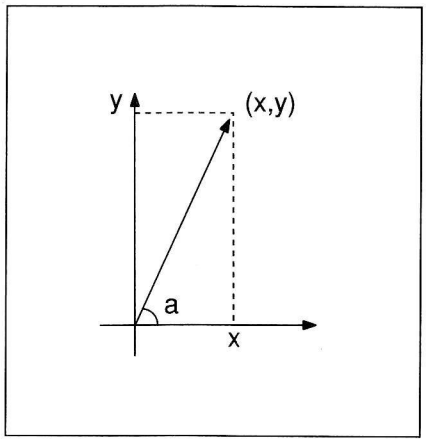


Figure 3: *The basic elements*

Hyperbolic Function	Circular Function
$\cosh x = \frac{e^x + e^{-x}}{2}$	$\cos x = \frac{e^{ix} + e^{-ix}}{2}$
$\sinh x = \frac{e^x - e^{-x}}{2}$	$\sin x = \frac{e^{ix} - e^{-ix}}{2i}$
$H_a = \begin{bmatrix}\cosh a & \sinh a \\ \sinh a & \cosh a\end{bmatrix}$	$R_a = \begin{bmatrix}\cos a & -\sin a \\ \sin a & \cos a\end{bmatrix}$
$H_a H_b = H_{a+b}$	$R_a R_b = R_{a+b}$
$e^x = \cosh x + \sinh x$	
$\ln x = 2\tanh^{-1}\frac{x-1}{x+1}$	

Table 1: *Hyperbolic functions*

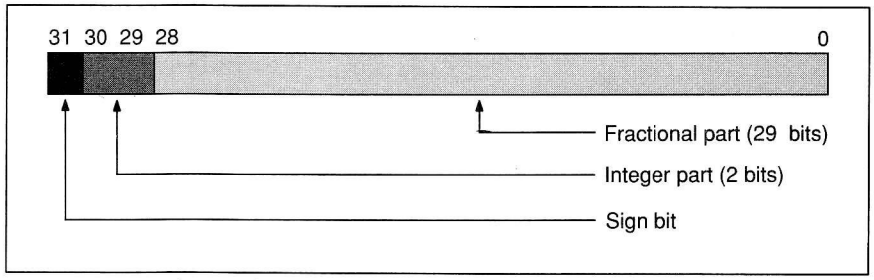


Figure 4: *Fixed point format*

(continued from page 154)
page 157, shows this in detail.

For hyperbolic functions, we start with $[x,y,z]$ and then drive z to zero. This yields the quantities in Example 4(b). Starting with $[x,y,z]$ and then driving y to zero gives the quantity shown in Example 4(c). For hyperbolics, $K \approx 1.21$.

Some interesting special cases include the exponential, square root, and natural logarithm. The exponential case is in Example 4(d) while the square root and logarithm cases are in Example 4(e).

Calculating the Constants

The algorithm to compute the circular and hyperbolic functions requires several precomputed constants. These include the scaling constant, K, for both circular and hyperbolic functions, and the sets shown in Example 5(a) and 5(b), respectively. Listing One illustrates this.

The program, written in C, uses fixed point arithmetic for all calculations. All constants and variables used to calculate functions are declared as the type *long*. The code assumes that a *long* is at least 32 bits. I have decided to represent numbers in the range $-4 \le x < 4$; this lets me represent e as a fixed point number. The high order bit is for the sign. The low order *fractionBits* (a constant defined as 29) bits hold the fractional part of the number. The remainder of the bits between the sign bit and the fractional part hold the integer part of the number. Figure 4 shows the fixed point format in graphic form.

I use power series to calculate the incremental angles a_i, as shown in Example 5(c) and 5(d), respectively. How do we know the number of terms necessary to evaluate $\tan^{-1}$ and $\tanh^{-1}$ to 32 bits of precision? First consider the value of x for which $\tan^{-1} x = x$ to 32 bits of precision. A theorem of numerical analysis states that for an alternating sum where the absolute values of the terms decrease monotonically, the error is less than the absolute value of the first neglected term. Solving the equation $x^3/3 = 2^{-32}$ for x yields $x = \sqrt[3]{6 \cdot 2^{-11}}$; therefore for $i \ge 11$, $\tan^{-1} 2^{-i} = 2^{-i}$ with 32 bits of precision.

For the higher powers of two, we need to solve the relation $2^{-in}/n = 2^{-32}$ for n for each of the cases $i = 1$ to 10. We do not even attempt the calculation for $i = 0$. The series for $\tan^{-1} 1$ converges very slowly, even after 500 terms the third digit is still changing. Fortunately, we know that the answer is $\pi/4$. Computing the rest is not as much work. The array *terms* has the gory details.

As usual, $\tanh^{-1}$ is more perverse. It is not an alternating sum and does not meet the conditions of the theorem used above. Consider the second neglected term of $\tanh^{-1} 1/2$. It is less than $1/4$ of the first neglected term because the series includes only every other power of two. All of the other neglected terms can have no effect on the 33rd bit. The series for the other arguments, $1/4, 1/8, \ldots$, converges even faster. Therefore, the number of terms calculated for $\tan^{-1}$ works just as well for $\tanh^{-1}$ for 32-bit accuracy.

Before computing the power series, we still need to compute the coefficients, $1/k$, for each term $k = 1, 3, 5, \ldots 27$. We fill the coefficient array *long a[28]* with odd indices by calling the routine *Reciprocal*, which takes two arguments and returns a *long*. The first argument is the integer for the desired reciprocal. The second specifies the desired precision for the fractional part of the result. Reciprocal uses a simple as can be restoring division; it is the algorithm we all learned in grade school for long division. The elements of the array *a* with even indices get *0L* because there are no terms in the power series with even exponents.

Everything is ready to fill the arrays *atan[fractionBits+1]* and *atanh[fractionBits+1]*.

The routine *Poly2* evaluates the power series for the specified number of terms for the specified power of two using Horner's rule. The coefficients come from the array *a*, which we just carefully filled. Horner's rule is the recommended method for evaluating polynomials. A polynomial as in Example 5(e) can be rewritten as in Example 5(f). This simple recursive formula evaluates the polynomial with n multiplications and n additions. We compute the prescaling constants K by using the method explained above; in the program we call these *X0C* and *X0H*, for the circular and hyperbolic constants, respectively. Program output to this point is shown in Listing Two, page 158.

The routines *Circular*, *InvertCircular*, *Hyperbolic*, and *InvertHyperbolic* are the C implementations of the algorithms described above. They all take as arguments the initial values for $[x,y,z]$; they leave their results in the global variables *X*, *Y*, and *Z*. Considering their versatility and the wide range of functions they compute, these routines are compact and elegant!

References

1. R. Nave. "Implementation of Transcendental Functions on a Numerics Processor." *Microprocessing and Microprogramming*, vol. 11, num. 3 – 4, pp. 221 – 225, March – April 1983.

2. A. K. Yuen. "Intel's Floating-Point Processors." *Electro/88 Conference Record*, pp. 48/5/1 – 7, 1988.

3. J. E. Volder. "The Cordic Trigonometric Computing Technique." *IRE Transactions Electronic Computers*, vol. EC – 8, pp. 330 – 334, September 1959.

4. D. S. Cochran. "Algorithms and Accuracy in the HP-35." *Hewlett-Packard Journal*, pp. 10 – 11, June 1972.

5. J. S. Walther. "A Unified Algorithm for Elementary Functions." In *1971 Proceedings of the Joint Spring Computer Conference*, pp. 379 – 385, 1971.

DDJ

(Listings begin on page 157.)

Vote for your favorite feature/article.
Circle Reader Service **No. 8.**

Listing One *(Text begins on page 152.)*

```
/* cordicC.c -- J. Pitts Jarvis, III
 * cordicC.c computes CORDIC constants and exercises the basic algorithms.
 * Represents all numbers in fixed point notation.  1 bit sign,
 * longBits-1-n bit integral part, and n bit fractional part.  n=29 lets us
 * represent numbers in the interval [-4, 4) in 32 bit long.  Two's
 * complement arithmetic is operative here.
 */

#define fractionBits 29
#define longBits 32
#define One    (010000000000>>1)
#define HalfPi (014441766521>>1)

/* cordic algorithm identities for circular functions, starting with [x, y, z]
 *  and then
 *  driving z to 0 gives: [P*(x*cos(z)-y*sin(z)), P*(y*cos(z)+x*sin(z)), 0]
 *  driving y to 0 gives: [P*sqrt(x^2+y^2), 0, z+atan(y/x)]
 *  where K = 1/P = sqrt(1+1)* . . . *sqrt(1+(2^(-2*i)))
 *  special cases which compute interesting functions
 *  sin, cos      [K, 0, a] -> [cos(a), sin(a), 0]
 *  atan          [1, a, 0] -> [sqrt(1+a^2)/K, 0, atan(a)]
 *            [x, y, 0] -> [sqrt(x^2+y^2)/K, 0, atan(y/x)]
 * for hyperbolic functions, starting with [x, y, z] and then
 * driving z to 0 gives: [P*(x*cosh(z)+y*sinh(z)), P*(y*cosh(z)+x*sinh(z)), 0]
 * driving y to 0 gives: [P*sqrt(x^2-y^2), 0, z+atanh(y/x)]
 * where K = 1/P = sqrt(1-(1/2)^2)* . . . *sqrt(1-(2^(-2*i)))
 *  sinh, cosh    [K, 0, a] -> [cosh(a), sinh(a), 0]
 *  exponential   [K, K, a] -> [e^a, e^a, 0]
 *  atanh         [1, a, 0] -> [sqrt(1-a^2)/K, 0, atanh(a)]
 *                [x, y, 0] -> [sqrt(x^2-y^2)/K, 0, atanh(y/x)]
 *  ln            [a+1, a-1, 0] -> [2*sqrt(a)/K, 0, ln(a)/2]
 *  sqrt          [a+(K/2)^2, a-(K/2)^2, 0] -> [sqrt(a), 0, ln(a*(2/K)^2)/2]
 *  sqrt, ln      [a+(K/2)^2, a-(K/2)^2, -ln(K/2)] -> [sqrt(a), 0, ln(a)/2]
 * for linear functions, starting with [x, y, z] and then
 *  driving z to 0 gives: [x, y+x*z, 0]
 *  driving y to 0 gives: [x, 0, z+y/x]
 */

long X0C, X0H, X0R; /* seed for circular, hyperbolic, and square root */
long OneOverE, E;   /* the base of natural logarithms */
long HalfLnX0R;     /* constant used in simultanous sqrt, ln computation */

/* compute atan(x) and atanh(x) using infinite series
 *    atan(x) =  x - x^3/3 + x^5/5 - x^7/7 + . . . for x^2 < 1
 *    atanh(x) = x + x^3/3 + x^5/5 + x^7/7 + . . . for x^2 < 1
 * To calcuate these functions to 32 bits of precision, pick
 * terms[i] s.t. ((2^-i)^(terms[i]))/(terms[i]) < 2^-32
 * For x <= 2^(-11), atan(x) = atanh(x) = x with 32 bits of accuracy  */
unsigned terms[11]= {0, 27, 14, 9, 7, 5, 4, 4, 3, 3, 3};
static long a[28], atan[fractionBits+1], atanh[fractionBits+1], X, Y, Z;
#include <stdio.h>      /* putchar is a marco for some */

/* Delta is inefficient but pedagogical */
#define Delta(n, Z) (Z>=0) ? (n) : -(n)
#define abs(n) (n>=0) ? (n) : -(n)

/* Reciprocal, calculate reciprocol of n to k bits of precision
 *  a and r form integer and fractional parts of the dividend respectively  */
long
Reciprocal(n, k) unsigned n, k;
{
  unsigned i, a= 1; long r= 0;
  for (i= 0; i<=k; ++i) {r += r; if (a>=n) {r += 1; a -= n;}; a += a;}
  return(a>=n? r+1 : r); /* round result */
}

/* ScaledReciprocal, n comes in funny fixed point fraction representation */
long
ScaledReciprocal(n, k) long n; unsigned k;
{
  long a, r=0; unsigned i;
  a= 1L<<k;
  for (i=0; i<=k; ++i) {r += r; if (a>=n) {r += 1; a -= n;}; a += a;};
  return(a>=n? r+1 : r); /* round result */
}

/* Poly2 calculates polynomial where the variable is an integral power of 2,
 *      log is the power of 2 of the variable
 *      n is the order of the polynomial
 *      coefficients are in the array a[]   */
long
Poly2(log, n) int log; unsigned n;
{
  long r=0; int i;
  for (i=n; i>=0; --i) r= (log<0? r>>-log : r<<log)+a[i];
  return(r);
}
WriteFraction(n) long n;
{
  unsigned short i, low, digit; unsigned long k;
  putchar(n < 0 ? '-' : ' '); n = abs(n);
  putchar((n>>fractionBits) + '0'); putchar('.');
  low = k = n << (longBits-fractionBits); /* align octal point at left */
  k   >>=  4;   /* shift to make room for a decimal digit */
  for (i=1; i<=8; ++i)
  {
    digit = (k *= 10L) >> (longBits-4);
    low = (low & 0xf) * 10;
    k += ((unsigned long) (low>>4)) - ((unsigned long) digit << (longBits-4));
    putchar(digit+'0');
  }
}
WriteRegisters()
{
  printf("  X: "); WriteVarious(X);
  printf("  Y: "); WriteVarious(Y);
  printf("  Z: "); WriteVarious(Z);
}
WriteVarious(n) long n;
{
  WriteFraction(n); printf("  0x%08lx 0%011lo\n", n, n);
}
Circular(x, y, z) long x, y, z;
{
  int i;
  X = x; Y = y; Z = z;
  for (i=0; i<=fractionBits; ++i)
  {
    x= X>>i; y= Y>>i; z= atan[i];
    X -= Delta(y, Z);
    Y += Delta(x, Z);
    Z -= Delta(z, Z);
  }
}
InvertCircular(x, y, z) long x, y, z;
{
  int i;
  X = x; Y = y; Z = z;
  for (i=0; i<=fractionBits; ++i)
  {
    x= X>>i; y= Y>>i; z= atan[i];
    X -= Delta(y, -Y);
    Z -= Delta(z, -Y);
    Y += Delta(x, -Y);
  }
```

(continued on page 158)

Listing One *(Listing continued, text begins on page 152.)*

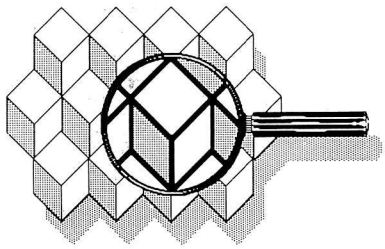
```
}
Hyperbolic(x, y, z) long x, y, z;
{
  int i;
  X = x; Y = y; Z = z;
  for (i=1; i<=fractionBits; ++i)
  {
    x= X>>i; y= Y>>i; z= atanh[i];
    X += Delta(y, Z);
    Y += Delta(x, Z);
    Z -= Delta(z, Z);
    if ((i==4)||(i==13))
    {
      x= X>>i; y= Y>>i; z= atanh[i];
      X  +=  Delta(y, Z);
      Y += Delta(x, Z);
      Z  -=  Delta(z, Z);
    }
  }
}
InvertHyperbolic(x, y, z) long x, y, z;
{
  int i;
  X = x; Y = y; Z = z;
  for (i=1; i<=fractionBits; ++i)
  {
    x= X>>i; y= Y>>i; z= atanh[i];
    X += Delta(y, -Y);
    Z -= Delta(z, -Y);
    Y += Delta(x, -Y);
    if ((i==4)||(i==13))
    {
      x= X>>i; y= Y>>i; z= atanh[i];
      X += Delta(y, -Y);
      Z -= Delta(z, -Y);
      Y += Delta(x, -Y);
    }
  }
}
Linear(x, y, z) long x, y, z;
{
  int i;
  X = x; Y = y; Z = z; z= One;
  for (i=1; i<=fractionBits; ++i)
  {
    x  >>= 1; z  >>= 1; Y += Delta(x, Z); Z -= Delta(z, Z);
  }
}
InvertLinear(x, y, z) long x, y, z;
{
  int i;
  X = x; Y = y; Z = z; z= One;
  for (i=1; i<=fractionBits; ++i)
  {
     Z -= Delta(z  >>= 1, -Y); Y += Delta(x  >>= 1, -Y);
  }
}

/*********************************************************/
main()
{
  int i; long r;
  /*system("date");*//* time stamp the log for UNIX systems */
  for (i=0; i<=13; ++i)
  {
    a[2*i]= 0; a[2*i+1]= Reciprocal(2*i+1, fractionBits);
  }
  for (i=0; i<=10; ++i) atanh[i]= Poly2(-i, terms[i]);
  atan[0]= HalfPi/2; /* atan(2^0)= pi/4 */
  for (i=1; i<=7; ++i) a[4*i-1]= -a[4*i-1];
  for (i=1; i<=10; ++i) atan[i]= Poly2(-i, terms[i]);
  for (i=11; i<=fractionBits; ++i) atan[i]= atanh[i]= 1L<<(fractionBits-i);
  printf("\natanh(2^-n)\n");
  for (i=1; i<=10; ++i){printf("%2d ", i); WriteVarious(atanh[i]);}
  r= 0;
  for (i=1; i<=fractionBits; ++i)
     r +=  atanh[i];
  r +=  atanh[4]+atanh[13];
  printf("radius of convergence"); WriteFraction(r);
  printf("\n\natan(2^-n)\n");
  for (i=0; i<=10; ++i){printf("%2d ", i); WriteVarious(atan[i]);}
  r= 0; for (i=0; i<=fractionBits; ++i) r +=  atan[i];
  printf("radius of convergence"); WriteFraction(r);

  /* all the results reported in the printfs are calculated with my HP-41C */
  printf("\n\n--------------------circular functions--------------------\n");
    printf("Grinding on [1, 0, 0]\n");
    Circular(One, 0L, 0L); WriteRegisters();
    printf("\n  K: "); WriteVarious(X0C= ScaledReciprocal(X, fractionBits));
    printf("\nGrinding on [K, 0, 0]\n");
    Circular(X0C, 0L, 0L); WriteRegisters();
    printf("\nGrinding on [K, 0, pi/6] -> [0.86602540, 0.50000000, 0]\n");
    Circular(X0C, 0L, HalfPi/3L); WriteRegisters();
    printf("\nGrinding on [K, 0, pi/4] -> [0.70710678, 0.70710678, 0]\n");
    Circular(X0C, 0L, HalfPi/2L); WriteRegisters();
    printf("\nGrinding on [K, 0, pi/3] -> [0.50000000, 0.86602540, 0]\n");
    Circular(X0C, 0L, 2L*(HalfPi/3L)); WriteRegisters();
  printf("\n------Inverse functions------\n");
    printf("Grinding on [1, 0, 0]\n");
    InvertCircular(One, 0L, 0L); WriteRegisters();
    printf("\nGrinding on [1, 1/2, 0] -> [1.84113394, 0, 0.46364761]\n");
    InvertCircular(One, One/2L, 0L); WriteRegisters();
    printf("\nGrinding on [2, 1, 0] -> [3.68226788, 0, 0.46364761]\n");
    InvertCircular(One*2L, One, 0L); WriteRegisters();
    printf("\nGrinding on [1, 5/8, 0] -> [1.94193815, 0, 0.55859932]\n");
    InvertCircular(One, 5L*(One/8L), 0L); WriteRegisters();
    printf("\nGrinding on [1, 1, 0] -> [2.32887069, 0, 0.78539816]\n");
    InvertCircular(One, One, 0L); WriteRegisters();
  printf("\n--------------------hyperbolic functions--------------------\n");
    printf("Grinding on [1, 0, 0]\n");
    Hyperbolic(One, 0L, 0L); WriteRegisters();
    printf("\n  K: "); WriteVarious(X0H= ScaledReciprocal(X, fractionBits));
    printf("  R: "); X0R= X0H>>1; Linear(X0R, 0L, X0R); WriteVarious(X0R= Y);
    printf("\nGrinding on [K, 0, 0]\n");
    Hyperbolic(X0H, 0L, 0L); WriteRegisters();
    printf("\nGrinding on [K, 0, 1] -> [1.54308064, 1.17520119, 0]\n");
    Hyperbolic(X0H, 0L, One); WriteRegisters();
    printf("\nGrinding on [K, K, -1] -> [0.36787944, 0.36787944, 0]\n");
    Hyperbolic(X0H, X0H, -One); WriteRegisters();
    OneOverE = X; /* save value ln(1/e) = -1 */
    printf("\nGrinding on [K, K, 1] -> [2.71828183, 2.71828183, 0]\n");
    Hyperbolic(X0H, X0H, One); WriteRegisters();
    E = X; /* save value ln(e) = 1 */
  printf("\n------Inverse functions------\n");
    printf("Grinding on [1, 0, 0]\n");
    InvertHyperbolic(One, 0L, 0L); WriteRegisters();
    printf("\nGrinding on [1/e + 1, 1/e - 1, 0] -> [1.00460806, 0,
                                                    -0.50000000]\n");
    InvertHyperbolic(OneOverE+One,OneOverE-One, 0L); WriteRegisters();
    printf("\nGrinding on [e + 1, e - 1, 0] -> [2.73080784, 0, 0.50000000]\n");
    InvertHyperbolic(E+One, E-One, 0L); WriteRegisters();
    printf("\nGrinding on (1/2)*ln(3) -> [0.71720703, 0, 0.54930614]\n");
    InvertHyperbolic(One, One/2L, 0L); WriteRegisters();
    printf("\nGrinding on [3/2, -1/2, 0] -> [1.17119417, 0, -0.34657359]\n");
    InvertHyperbolic(One+(One/2L), -(One/2L), 0L); WriteRegisters();
    printf("\nGrinding on sqrt(1/2) -> [0.70710678, 0, 0.15802389]\n");
    InvertHyperbolic(One/2L+X0R, One/2L-X0R, 0L); WriteRegisters();
    printf("\nGrinding on sqrt(1) -> [1.00000000, 0, 0.50449748]\n");
    InvertHyperbolic(One+X0R, One-X0R, 0L); WriteRegisters();
    HalfLnX0R = Z;
    printf("\nGrinding on sqrt(2) -> [1.41421356, 0, 0.85117107]\n");
    InvertHyperbolic(One*2L+X0R, One*2L-X0R, 0L); WriteRegisters();
    printf("\nGrinding on sqrt(1/2), ln(1/2)/2 -> [0.70710678, 0,
                                                    -0.34657359]\n");
    InvertHyperbolic(One/2L+X0R, One/2L-X0R, -HalfLnX0R); WriteRegisters();
    printf("\nGrinding on sqrt(3)/2, ln(3/4)/2 -> [0.86602540, 0,
                                                    -0.14384104]\n");
    InvertHyperbolic((3L*One/4L)+X0R, (3L*One/4L)-X0R, -HalfLnX0R);
    WriteRegisters();
    printf("\nGrinding on sqrt(2), ln(2)/2 -> [1.41421356, 0, 0.34657359]\n");
    InvertHyperbolic(One*2L+X0R, One*2L-X0R, -HalfLnX0R);
    WriteRegisters();
    exit(0);
}
```

End Listing One

Listing Two

```
atanh (2^-n)

1   0.54930614   0x1193ea7a 002144765172
2   0.25541281   0x082c577d 001013053575
3   0.12565721   0x04056247 000401261107
4   0.06258157   0x0200ab11 000200125421
5   0.03126017   0x01001558 000100012530
6   0.01562627   0x008002aa 000040001252
7   0.00781265   0x00400055 000020000125
8   0.00390626   0x0020000a 000010000012
9   0.00195312   0x00100001 000004000001
10  0.00097656   0x00080000 000002000000

radius of convergence 1.11817300

atan (26-n)

0   0.78539816   0x1921fb54 003110375524
1   0.46364760   0x0ed63382 001665431602
2   0.24497866   0x07d6dd7e 000765556576
3   0.12435499   0x03fab753 000376533523
4   0.06241880   0x01ff55bb 000177652673
5   0.03123983   0x00ffeaad 000077765255
6   0.01562372   0x007ffd55 000037776525
7   0.00781233   0x003fffaa 000017777652
8   0.00390622   0x001ffff5 000007777765
9   0.00195312   0x000ffffe 000003777776
10  0.00097656   0x0007ffff 000001777777

radius of convergence 1.74328660

------- circular functions ------------------


K: 0.60725293       0x136e9db3     002333516663


------- hyperbolic functions ----------------


K: 1.20749708       0x26a3d0ed     004650750355

R: 0.36451229       0xbaa15b1      001352412661
```

End Listings



Memory Commander, recently announced by **V Communications,** provides up to 960K of contiguous DOS memory, maps BIOS ROMs into faster RAM, has LIM 4.0 EMS emulation, and loads TSRs and device drivers into high memory. With Memory Commander you can install networks, drivers, TSRs, and large applications simultaneously.

DDJ spoke with Rick Gilligan of Computer and Software Enterprises Inc., who runs MS-DOS 4.01 to take better advantage of his 143-Mbyte disk drive. He said that "with Memory Commander I can move device drivers into high memory and run CAD programs that require as much as 550K of low memory. I can have one configuration now that gives me more than enough memory, and I can still have buffers, files, and drivers for my programs. Memory Commander takes care of a lot of things you would have to use separate drivers for; it includes EMS, VCPI, XMS, and ANSI drivers. The only conflict is with Windows 3.0, but no DOS extender works with Windows in protected mode."

A "control panel" describes how memory is utilized and provides control options for customizing Memory Commander for your system. A built-in RAM disk and an internal ANSI.SYS replacement are included for extended screen and keyboard support, and these additions require no DOS memory. The product sells for $129.95. Reader service no. 20.
V Communications Inc.
4320 Stevens Creek Blvd., Ste. 275
San Jose, CA 95129
408-296-4224
800-648-8266

ROM-DOS, a special-purpose ROMable DOS for non-PC applications, is being offered by **Datalight**. This small, modifiable operating system provides the functionality of MS-DOS 3.2, can operate from within ROM, and can boot from a floppy or hard disk. It can run MS-DOS executable COM and EXE files generated by any MS-DOS compiler.

Kevin Kyriss of 2Morrow, makers of a hand-held computer called "Silent Partner," told *DDJ* that their product had a limited future until Datalight put their operating system on it and "opened it up to a tremendous amount of applications. We can't have a disk drive; we have RAM that is write-protectable, and ROM-DOS goes into RAM and looks like ROM. You can execute applications directly out of ROM, which is more efficient for RAM."

You can write applications with the language of your choice, compile it with whatever compiler you choose, and then load it into ROM along with ROM-DOS to run in the target system with no other modifications. With all functions included, ROM-DOS takes about 34K of ROM and uses as little as 14K of RAM when running. And ROM-DOS can run applications directly, conserving the RAM required for COMMAND.COM.

Datalight has developed a mini-BIOS for use in embedded systems, which provides support for a remote console, hardware timer, and serial ports. ROM-DOS can be customized to include only those functions that are used by a particular application. A ROM-DOS Developer's Kit is available for $495, and a license to the source code costs $10,000. Reader service no. 23.
Datalight
17505 68th Ave. NE, Ste. 304
Bothell, WA 98011
206-486-8086

A C code generator for Borland's Paradox Engine is available from **Concept Dynamics**. PARAGen examines the data structure of user-defined Paradox tables and produces code to perform a variety of operations on those tables.

Daniel Sullivan of Concept Dynamics told *DDJ* that "we developed PARAGen for our own internal use. We gave a demo to a Paradox user's group, and found lots of interest in it. We are trying to appeal to a cross-over segment, to C programmers who are interested in database development but haven't used Paradox because of the prior inability to read and write Paradox files. Our next version will support C++ and Pascal, as well as C."

PARAGen allows developers to use the PC to generate database access code specific to their applications. The Paradox Engine provides generic functions for performing table operations but these have no knowledge of the tables defined by the application developer — PARAGen links an application with the Paradox Engine.

Functions generated by PARAGen will open, close, create, and empty Paradox tables. Record-level operations include full and partial key searches, first, last, next, and previous record fetches, and field searches. Append, insert, and delete record operations are also provided. The product sells for $99, and demonstration versions are available on CompuServe. Reader service no. 21.
Concept Dynamics Ltd.
1147 S. Euclid
Oak Park, IL 60304
708-524-2814

Version 2.1 of TSX-32, a multiuser, multitasking, and DOS-compatible operating system for 80386 and 80486 machines, has been announced by **S&H Computer Systems.** TSX-32 offers virtual memory with demand paging, file access control, printer spooling and queueing, and 32-bit program execution. It is binary-compatible with DOS programs that use Phar Lap's DOS Extender.

Some new features are faster I/0 (achieved through improvements in data caching, cache flushing, and desk seek optimization), VGA and EGA color graphics, SCSI host adapters, and 9-track tape. The system uses an "Adaptive Scheduling Algorithm" (ASA) that dynamically orders and reorders system priorities based on the nature of the tasks to be performed. The ASA makes distinctions between interactive, real-time, and low-priority operations. An unlimited users license with one year of support and updates costs $1,450. Reader service no. 32.
S&H Computer Systems Inc.
1027 17th Ave. South
Nashville, TN 37212
615-327-3670

Microsoft Windows developers may be interested in **JT Software's** JTW C++ class library, which provides a ready-made environment for developing Windows applications.

JTW includes more than 96 classes, a portion of which act as a high-level interface between Windows and the application (to facilitate ports to other environments). Other classes provide generic means for representing the application's data, either in the form of simple container classes, or in a higher-level family of classes that integrate into the windowing environment by keeping different views of the data in sync, and by providing members that generically handle common operations such as those required by a standard File menu.

Additional classes represent com-

(continued on page 166)

(continued from page 161)
monly used windows, dialog boxes, and controls (TextEdit, OpenDialog, PushButton). And procedural and object-oriented graphics capability is included. The $150 price includes full sources, user guide and reference manual, and sample programs. JTW supports the Zortech and Glockenspiel C++ compilers and requires Windows SDK. Reader service no. 22.
JT Software
P.O. Box 4292
Santa Clara, CA 95054
408-727-8591

SilverScreen, a 3-D CAD/Solids Modeling software system from **Schroff Development Corporation,** is now available for licensing as a CAD engine. Some of SilverScreen's features are 2-D and 3-D Boolean operations, shading (and shadows), rendering, hidden line removal, camera walk, text editor, associative dimensioning, mass properties, the ability to import and export DXF and IGES files, and the ability to export in the HPGL format.

The system includes a resident C compiler that allows you to create custom applications. The compiler implements a library of over 200 functions, including a subset of the standard C library. And developers have access to a large number of functions which use the SilverScreen database and control the SilverScreen environment. Contact the company for pricing information. Reader service no. 24.
Schroff Development Corp.
4732 Reinhardt Dr.
Roeland Park, KS 66205
913-262-2664

GrafPrint, printer graphics libraries that support Turbo C/C++, Microsoft C/QuickC, and Watcom C and C/386 graphics libraries, are available from **AnSoft Inc**. You can develop programs in any video mode, and GrafPrint maps the screen graphics to any other video mode and to the printer.

It currently supports output on HP LaserJet/DeskJet/PaintJet, Epson FX/MX/LQ, and compatibles. Grafprint can use EMM 3.2 and over and adds integer and floating-point viewports on the screen and printer. GrafPrint Personal sells for $150 and GrafPrint Developers for $300 (no royalties). Reader service no. 31.
AnSoft Inc.
8254 Stone Trail Ct.
Laurel, MD 20723
301- 470-2335

A fully supported library of reusable C++ software components is available from **Empathy Inc.** Classix incorporates object-oriented design techniques and heuristics in C++ classes that cover abstractions in the areas of data structure design, mathematical support objects, Smalltalk-like classes, and mimics of the primitive data types. Classix also includes a parameterization utility for reusing classes with different data types.

Classix has over 35 classes and 800 operations, and includes inheritance and containment in the underlying design structure. Classix is offered in source code format only for use with Glockenspiel C++, Sun C++, GNU C++, Zortech C++, and MPW C++. Prices start at $295. Reader service no. 25.
Empathy Inc.
P.O. Box 632
Cambridge, MA 02142
617-787-3089

NABJA Software, developer of object-oriented programming tools, has announced that NABJAooc, its object-oriented development environment for C, now supports Microsoft C 6.0.

NABJAooc uses MS C's help system to provide an online hypertext reference that includes a full class hierarchy which allows the user to browse through both system and user class definitions. It was designed for C programmers who want to code in ANSI C but also want the benefits of OOP — without learning a new language.

NABJAooc features full implementation of the object-oriented paradigm, including multiple inheritance; support for integrated programming environments; compatibility with existing source-level debuggers, including CodeView; foundation classes, with source code, for basic data structures; a message-trace facility; and less than 10K of overhead for the base system. NABJAooc sells for $49, or $99 with source code. Reader service no. 27.
NABJA Software
P.O. Box 413
Girard, PA 16417-0413
814-774-3699

Baran's Tech Letter, a publication covering the NeXT Computer, provides news and analysis of new products and technological developments for and around the NeXT Computer.

Nicholas Baran told *DDJ* that "there are no publications for the NeXT Computer other than by user groups and an academic report published by the company" and, said Baran, "there's a need for a publication that's not in love with Steve Jobs."

Baran's Tech Letter will cover such topics as third-party software, Postscript Level 2, Motorola's 96002 digital signal processor, and the level of dedication of such players as IBM. IBM has licensed NextStep, and according to Baran appears to be hedging its bets on whether NextStep or OSF Motif will emerge dominant. The subscription rate for 12 issues is $125. Reader service no. 28.
Baran's Tech Letter
P.O. Box 876
Sandpoint, ID 83864-0876
208-265-5286

Pixelab has released GSPOT, a symbolic debugger for the Texas Instruments 34010. GSPOT (short for "graphics system processor operating tool") includes full floating-point support and TIGA support, and is designed to run on a PC or compatible. Its features are designed to make debugging in a "host-target" environment simple; for example, the ability to install GSPOT as a resident program while using another debugger for the host processor.

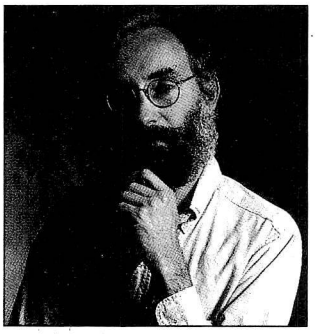
GSPOT can be configured to work with any GSP-based hardware, including TI's SDB board. Other features include symbolic debugging, C source debugging, a user-interface similar to CodeView, online assembly, a memory display/edit mode, watchpoints, script files, and so on. Version 1.0 sells for $995. Reader service no. 30.
Pixelab Inc.
4513 Lincoln Ave., Ste. 105
Lisle, IL 60532
708-960-9339

ExploreNet 3000, a neural network application development environment that uses Windows 3.0, has been announced by **HNC**. A variety of predefined example applications are included. ExploreNet 3000 is a flexible, icon-based environment in which users can define and create applications by selecting icons on the screen, specifying parameters, and linking individual modules together. Some major modules are the file module, data transformation module, pipe module, network module, and display module.

ExploreNet 3000 also incorporates watchpoints and control functions for monitoring network training and testing results. A programmatic interface is provided if you prefer to work in C. The product sells for $1,495. Reader service no. 29.
HNC
5501 Oberlin Dr.
San Diego, CA 92121-1718
619-452-6524

DDJ

Is Copyrighting Software Futile?



The purpose in copyrighting a piece of software is to control its copying and/or commercial use. Richard Stallman has extensively questioned the desirability of this purpose. I'm going to do something simpler: Question its feasibility. Let's consider three cases.

Case I: Copying at the Office

Laws that cannot be enforced are vacuous. Laws whose violations cannot be detected cannot be enforced. Violations occurring in an office cannot be detected unless an employee blows the whistle or the violator is careless. Whistle-blowing is not a common phenomenon. Consider conscientious office worker Clara Tillinghast. Her company pollutes the environment, at least one male executive she knows of makes improper advances to female employees, her boss regularly lies to customers, and the person in the next office has an unauthorized copy of Lotus 1-2-3. What will Clara do?

Even if Clara doesn't get on the phone to Lotus, the person in the next office might get careless, leaving a hand-labeled diskette lying about when Jim Manzi is visiting the office, say. But with the prevalence of hard disks, there need be no illegal diskette to observe, and the illegal copy can be erased in seconds. No, the sad fact is that user carelessness is not something software vendors can count on.

One strategy is to issue site license contracts that are cheap enough to make even the small risk of detection unacceptable. But this leaves pricing out of vendor control. Since there is no clear formula for determining the ratio of site licensing price to single copy price, tight competition in any product category could conceivably drive this ratio arbitrarily close to one. That is, the vendor will effectively sell one copy per office.

But it has been mooted that what really sells multiple copies of a program into an office is the need for manuals. If so, the trend toward online documentation could be disastrous. And the trend is compelling: The user really ought to be able to get the answer at the point and in the medium where the question arises; software can be produced more economically without hardcopy manuals; and it can be produced to tighter deadlines, hence more competitively, if two distinct production lines don't have to finish simultaneously. If hardcopy manuals are all that makes multiple sales to an office possible, the elimination of hardcopy manuals will also lead to one copy per office.

Case II: Copying at Home

Here the risk of detection due to carelessness and the reward for whistle blowing both drop to zero, which is why copy protection hasn't gone away in game software. But games are generally cheap and ephemeral, and copy protection has been so discredited in the broad business software market that it is unlikely to come back. What happens if the trend toward working at home continues?

Consider the pressures. The company supplies ambitious home-office worker Bob Slocum with mediocre spreadsheet program *X*. A barrage of ads tells Bob that spreadsheet program *W* will give him the edge over his rival for promotion, Jack Green. Bob can't afford to buy *W* himself and the company won't spring for it, but he's sure that Jack is just the sort of guy to take unfair advantage by pirating *W*. As a home-office worker, Bob naturally belongs to a user group, where he knows he can pick up *W* for nothing. What will Bob do?

It's easy to imagine a vicious cycle developing, with home-office workers copying software because they can't afford to buy it and vendors raising prices because of the diminished sales. Pretty soon, they're selling one copy per user group.

Case III: Copying in Cyberspace

Let's postulate what many would call the worst scenario, and what I have been painting as the most plausible scenario. No office buys more than one copy of a piece of software, and all home users get their software from friends in user groups. This still lets the software vendor sell a copy to each office and each user group, right?

It's actually possible to imagine an industry configured this way, although pricing for software would have to be high because every purchase would be a group purchase. In this scenario, the direct customers would be defined by access considerations. Since copyright violation would still be illegal, sending unauthorized copies through the mail might be seen as too risky, and sending them by phone would surely be too expensive. Copying would all be local: User groups would be sneakernet-bound and offices would be site-bound. User groups would have to find a way to get money to pay for the software they'd buy, but selling support suggests itself as a possibility. It might work.

Maybe. We'll have to see what the bit-per-second pricing for phone service is when wide-bandwidth phone service goes national in 1992.

Michael Swaine
editor-at-large

#170 NOVEMBER 1990 $3.50 ($4.50 CANADA)

15 YEARS OF POWER PROGRAMMING

Dr. Dobb's JOURNAL

SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

ROLLING YOUR OWN OBJECT-ORIENTED LANGUAGE



PLUS

- C++
- TURBO PASCAL
- SMALLTALK
- ACTOR

ARE SOFTWARE PATENTS *KILLING* INNOVATION ?

DEBUGGING MACINTOSH APPS

DEVELOPING FOR WINDOWS 3.0

AND MORE ON

- DATA ENCRYPTION
- DOS EXTENDERS

0 38351 16562 8 11

CONTENTS

NOVEMBER 1990
VOLUME 15, ISSUE 11

FEATURES

ROLL YOUR OWN OBJECT-ORIENTED LANGUAGE 16
by Michael Floyd
Mike defines, designs, and implements an object-oriented language that you can wrap around your code.

AN EXISTENTIAL DICTIONARY 20
by Edwin T. Floyd
You can avoid the overhead of conventional search techniques by recording the existence of a key — without storing the key itself.

OBJECT-ORIENTED DEBUGGING 36
by Simon Tooke
Simon examines strategies and tools for object-oriented debugging, using C++ as an example.

CTRACE: A MESSAGE LOGGING CLASS 44
by William D. Cramer
Augment your Macintosh development environment with this general-purpose message logging window that provides basic *printf( )* capabilities.

SOFTWARE PATENTS 56
by The League for Programming Freedom
Will software patents kill innovation in the software development field? Here's one view.

ROLL YOUR OWN DOS EXTENDER: PART II 74
by Al Williams
Al covers debugging and 80386 exceptions and takes you under his DOS extender's hood.

THE MVC PARADIGM IN SMALLTALK/V 168
by Kenneth E. Ayers
In Smalltalk/V, *MVC* is spelled *OPD*. Ken examines both the Model-View-Controller and the Object-Pane-Dispatcher.

EXAMINING ROOM

PROGRAMMER TOOLS FOR ACTOR 3.0 86
by Marty Franz
The Windows 3.0 surge spurred Marty to take a look at Actor 3.0 and two of its support tools — WinTrieve and the Whitewater Resource Toolkit.

PROGRAMMER'S WORKBENCH

WINDOWS 3.0 APPLICATION DEVELOPMENT 92
by Walter Knowles
Developing sophisticated Windows 3.0 applications requires a variety of tools. Walt uses ToolBook, db_VISTA, and a C compiler to build a checkbook accounting application.

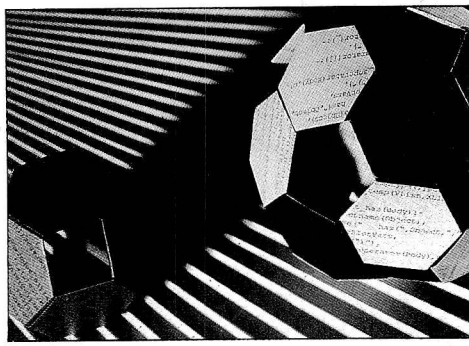
COLUMNS

PROGRAMMING PARADIGMS 141
by Michael Swaine
Michael scrutinizes the *New Connectionism*, an emerging movement in cognitive and computer science that involves both neural nets and parallel distributed processing.

C PROGRAMMING 149
by Al Stevens
Al returns to the data encryption algorithm and the DES discussion he began in September.

STRUCTURED PROGRAMMING 155
by Jeff Duntemann
Is it Turbo Pascal's modulus operator that's been giving Jeff's day-of-the-week function fits? What would Zeller have to say about that!

PROGRAMMER'S BOOKSHELF 161
by Andrew Schulman
Andrew looks at two books by object-oriented pioneers — *Object-Oriented Design with Applications* by Grady Booch and *Object-Oriented Software Construction* by Bertrand Meyer.



The geometric shapes on the cover are examples of a new style of paperfolding called "modular origami." Developed in Japan by Tomoko Fuse, squares of paper are folded into units that lock together in a puzzle-like fashion without the use of glue. The models in this magazine were folded by Vicky Mihara Avery.

DEPARTMENTS

EDITORIAL 6
by Jonathan Erickson

LETTERS 8
by you

SWAINE'S FLAMES 184
by Michael Swaine

PROGRAMMER'S SERVICES

ADVERTISER INDEX 176
where to go for more information on products

OF INTEREST 177
compiled by Janna Custer

PROGRAMMER'S MARKETPLACE 178
classified ads

SOURCE CODE AVAILABILITY
As a service to our readers, all source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (inside Calif.) or 800-533-4372 (outside Calif.). Specify issue number and disk format. Code is also available through the DDJ Forum on CompuServe (type *GO DDJ*) and through M&T's Telepath Conferencing system.

NEXT ISSUE

With the onset of winter, we'll examine the hard, cold facts surrounding data communications and connectivity in our December issue.

School Days, Legal Maze

It was in our July issue, if you recall, that I announced the Kent Porter Scholarship for college students working toward degrees in computer science. Now that the school year is underway, I'm honored to award scholarships (in the amount of $500 each) to the following individuals:

Victor J. Duvanenko
North Carolina State University
Raleigh, North Carolina

Theresa McMurray
Stephen F. Austin State University
Nacogdoches, Texas

Mark S. Kessel
University of Colorado/Boulder
Boulder, Colorado

James D. Marco
State University of New York
Utica, New York

Congratulations to each and every one. Those of us at *DDJ* are glad to play at least a small part in helping you realize your goals.

I'd also like to once again thank those of you who made contributions to the scholarship fund. And don't forget that because the program is on-going, we'll be awarding scholarships for the 1991-92 school year.

Software Patents

It's not often that I ask you to read a particular article, but this month is an exception. For goodness sakes, turn to page 64 and read, think about, and respond in some way to the article "Software Patents." If you write software, this analysis may be one of the most important articles you'll read for a long time to come.

Is the practice of patenting algorithms dangerous and, in the end, detrimental to the spirit of innovation in the software development process? Consider this recent phone call from a *DDJ* reader: The small software house where the caller worked had spent nearly a year developing an application for a client. The program did what it was supposed to do very well and the client accepted it. Upon examining the source code, however, one of the client's programmers ran across an algorithm he knew to be patented. Until that time, our caller had no knowledge of any existing patents on the algorithm. Furthermore, he said, he may very well have implemented several other patented algorithms — he just doesn't know. The algorithm in question is widely implemented, as were others in the same program. In any event, the client is now requiring licenses on all patented algorithms before accepting the package. The caller would be more than willing to pay any necessary license fees, if he just knew which algorithms were patented and who owns those patents. The first question you might ask is, what would you do if you were in the caller's shoes? After you read the article, don't stop there. Tell us what you think about the subject. Let us know if you've had problems dealing with patented programming techniques such as that described above. Or, for that matter, tell us if you hold a software patent and why you think it is important to do so.

Incidentally, the September/October 1990 issue of *American Heritage* magazine includes a historical perspective on how we got into the patenting mess we're in today. The article, entitled "The Power of Patents" by Oliver E. Allen, paints a not-so-pretty picture, and you can easily draw parallels between the patent process in the Industrial Age in the first part of this century and what's going on today. For further study, the author recommends *The United States Patent System: Legal and Economic Conflicts in American Patent History* by Floyd L. Vaughn (University of Oklahoma Press, 1956) and *America by Design* by David F. Noble (Alfred A. Knopf, 1977).

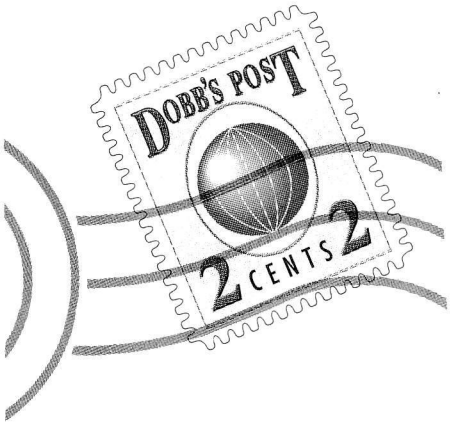
Rhealstone Lives!

Nearly two years ago, *DDJ* proposed the Rhealstone, a suite of benchmarks for evaluating real-time systems, authored by Robin Kar. (See "Rhealstone: A Real-Time Benchmarking Proposal," February 1989 and "Implementing the Rhealstone Real-Time Benchmark," April 1990.)

Our proposal was just that — a proposal — and we wanted others to evaluate, criticize, refine, and continue the dialogue. With this in mind, you might want to check out the article "Shootout at the RT Corral" by Bruce Koball and Alex Novickis in *Embedded Systems Programming* magazine (September 1990) where the authors analyze and polish the Rhealstone (and Hartstone) benchmarks.



Jonathan Erickson
editor-in-chief



Catching A Few Rays

Dear *DDJ*,

Your recent article, "Ray Tracing," by Daniel Lyke (September 1990) has touched on a subject close to my heart. We in the optics business are always tracing rays, not to make nice pictures, but to design lenses. My own specialty is stray light analysis, in which even more rays are traced. The equations used in the article to describe reflection in terms of a unit normal vector and an incident ray unit vector, are among the equations I use in writing stray light analysis programs. In stray light analysis programs, however, we also compute scattered and diffracted "rays" with good radiometric accuracy.

Let me back up for a moment and briefly describe what stray light analysis is. I'm sure you've heard the advice given to amateur photographers to avoid shooting into the sun. The reason is that the sun will reflect off of lens surfaces and scatter off the inside of the lens barrel, causing much stray light and spoiling the picture. The job of the stray light analyst is to quantify this stray light versus position of the sun, or any other bright source of light. Several computer programs have been written to accurately analyze stray light. The accuracy required is extreme: An optical system may reduce stray light by a factor of 10^{-10} with an elaborate system of baffles, and the answer must still be known within about a factor of two.

For the past ten years I have maintained a stray light analysis program (written in Fortran around 1970 for a mainframe computer), and I have recently completed a new stray light analysis program (under contract) written in C for the IBM PC. Both use Monte Carlo methods to sample the stray light. I have always viewed the computer graphics industry from afar, with envy, but long ago realized that stray light programs would make wonderful engines for creating photo-realistic images. Alas, I have never had the time (i.e. funding) to find out just how good they would be.

The programs I work with can currently model any second-order surface. By this I mean any surface that can be represented by a second-order equation in Cartesian coordinates. This includes spheres, cones, any conic section of revolution (paraboloids, ellipsoids, hyperboloids, oblate spheroids), and all combinations such as parabolic cylinders, elliptic cones, hyperbolic paraboloids, etc. They can also model planes and tori, which are fourth-order surfaces. Many objects can be built up from these basic surface types. Not only that, but stray light analysis programs can also accurately model light emitted by hot objects.

The aspect of your article that struck home is that the same techniques you used for tracing rays are also used in stray light analysis programs, with surfaces being represented in a global coordinate system and second-order ray-surface intercepts being solved using the good old quadratic equation. The rest of the optics industry represents surfaces relative to one another with no global coordinate system and uses specialized ray tracing equations. A difference between stray light and computer graphics is that with my programs, I often trace rays starting from the source and going toward the detector, just like the real photons. (The detector may be an eye, photographic film, vidicon, or electro-optic detector.) The computer graphics industry seems to have revived the ideas of the ancient Greeks, who believed that light emanates from the eyes to allow the observer to see.

In conclusion, I have nothing to report other than my own enthusiasm and hope that I may have time to experiment with computer graphics before it's too late. I am soon to have a 486 computer with 1024 VGA display on my desk, however, so I may find the temptation to experiment irresistible.

Edward R. Freniere, Vice President
Telic Optics, Inc.
Marlborough, Massachusetts

Optimizing Super VGA

Dear *DDJ*,

Christopher Howard's "Super VGA Programming" (*DDJ*, July 1990) is well written and informative. I have a problem, however, with his inference (at the top of page 22) that checking for a video bank switch must be done on a pixel-by-pixel rather than a line-by-line basis.

It is entirely possible to load not only a line, but multiple lines, without worrying about running into a bank boundary. The following code fragment (taken directly from Blackhawk's "Database Graphics Toolkit") illustrates how to do this (the numbers chosen let us load 80 columns of previously formatted 16-line graphics-mode text with a single move ($80 \times 8 = 640$; $640 \times 16 = 10240$; $65536 - 10240 = 55296$)). See Example 1.

The substance of this method is to divide the bank-switching problem into a video hardware component and a memory access component; it works because a memory reference falling within video memory will always be directed to video memory, whether or not the segment register points to its start.

A possible objection to the method is that, on a dual-monitor system, one might run into the monochrome video buffer at segment 0B000H, but, by shifting the DI register down, the method itself prevents this.

John D. Brink, President
Blackhawk Data Corporation
Chicago, Illinois

Chris responds: My quick comment about optimizations only stated that some optimizations *were limited. To expand a little further, I know of some software packages that optimize by storing an array of pointers to each scanline. This gets around interleaving problems, etc., but it does not work if the scanline crosses a 64K bank boundary.*

What you refer to in your letter is similar to what we do in our GX Development Series *of programming tools. We call it* Thresholding *which defines the point at which we can move a*

(continued on page 12)

```
bnkchk: push   dx;          AH:DI contains computed offset.
          cmp    ah, BNK;     Have we gone past a boundary?
          je     sambnk;      If so, no hardware changes.
          mov    BNK, ah;     Save new bank number.
          call   [BNKAD];     Do hardware bank switch.
sambnk: mov    dx,0a000h;   Load standard VidRam segment.
          cmp    di,55296;    Memory wraparound possible?
          jb     bnkext;      If not, use standard VidRam segment.
          add    dx,640       Move segment forward 640 paragraphs.
          sub    di,10240;    Move index back same amount.
bnkext: mov    es,dx;       Set ES register.
          pop    dx;          DX preserved, AX destroyed.
          ret;                ES,DI altered on return.
```

Example 1

(continued from page 8)
given amount of data the fastest way possible. If we are below the threshold, we use fast rep movsw *instructions. If we are above the threshold, we need to move more slowly and wait for the point at which the bank needs changing.*

However, your programming example is incorrect as written. When you are checking for a memory wraparound, you are in effect checking for what we call the threshold. Except if a wraparound is possible, you are simply calculating a new segment:offset *for the same memory location. This does not prevent the wraparound, it merely insures that DI does not overflow. In fact, it converts a memory wraparound problem into a memory overwrite problem. Instead of wrapping back up to A000H, it makes sure that data is written past the VGA video address space into B000H. You bring up this point yourself, but you mention that "the method itself prevents this." By your example, the method does not prevent this. You are correct that the segment does not have to point to the top of video memory (an address is an address — you can point to the same place with many* segment:offset *pairs), but that still does not help here.*

Perhaps in illustrating your point, some code was inadvertently left out of your example. In any case, you are correct in that it is possible to optimize around the 64K bank switching without testing every single pixel. The point I was making in the article was simply that some optimizations are not possible or are more difficult to implement if the bank ends in the middle of a scanline. Your example serves to illustrate this perfectly.

Throwing A Slow Curve

Dear *DDJ*,

Thanks for the interesting graphics issue (*DDJ*, July 1990). I really appreciate learning about algorithms that I can actually use. When I first read Todd King's article on Bézier curves, I was amazed at the time performance difference he found between the deCasteljau and literal rendering methods of drawing Bézier curves. But after a careful look at the listing, I realized that most of that difference is due to a difference in the number of lines drawn per Bézier curve by the methods, rather than differences in algorithmic efficiency. In King's program, for the literal rendering method, 101 lines per curve are drawn; for the deCasteljau method, either 24 (for cut-off 3) or 48 (for cut-off 4) are drawn. By changing the increment in the *for* loop for the literal method to 0.4 or 0.2, 25 or 51 lines per curve are drawn for the literal method with little or no loss in appearance. (Note: The article says that the variable DPU is used to determine the number of lines drawn for the literal method, but DPU is defined in the listing, then never used.) With this improvement, the deCasteljau method is only 2.8 times as fast, instead of five to ten times as fast.

The literal method is also unnecessarily slow in King's implementation because of inefficient coding. I replaced the routine *draw_bezier1* with the code in Example 2 to avoid recalculating reused factors. With this alteration, the deCasteljau method is only about twice as fast as the literal method. The differences are smaller if a floating-point coprocessor is available: deCasteljau is only about 1.2 times as fast as improved literal with an 80*x*87.

Further improvements can be made to both methods: 1) If a curve is really a straight line, it should be drawn as a line, not a Bézier curve; 2) When a character is scaled down, many of the lines in each curve have zero length on-screen, and don't need to be drawn; 3) General efficiency clean-ups.

The biggest improvement possible with no 80*x*87 is to use all integer and long calculations. I have used carefully scaled integers and the literal method in my shareware PostScript drawing program PictureThis, and have reduced the drawing time dramatically. To draw King's six *a*'s (with no fills) on an XT-clone with no 80*x*87 takes 20.5 seconds for the deCasteljau method (24 lines/curve), 34 seconds for the improved literal method (21 lines/curve), 47 seconds for the unimproved literal method (21 lines/curve), but only three seconds for PictureThis with an integer literal method (19 lines/curve). An integer deCasteljau method should make PictureThis slightly faster.

PictureThis provides kerning, accent characters, subscripts and superscripts, font modifications, yet it runs quite well even on a 4.77 MHz PC with 512K memory, CGA (including Hercules with CGA emulation), no expanded memory, no mouse, and no hard disk. PictureThis produces standard Encapsulated PostScript (EPS) files (*with* "showpages"!).

Pat Williams
Hot Ideas Publishing
Rt. 1, Box 302
Gravel Switch, KY 40328

Todd responds: I had also wondered about the speed difference between the literal and deCasteljau methods of calculating a Bézier curve. Admittedly, I should have taken a closer look at the code and I should have done the same type of analysis you did in order to make a more fair comparison. One other observation is that with either method, a curve is drawn as a series of line segments. The only difference is how the end points of the line segments are calculated. In the deCasteljau method the calculation involves 12 multiplication/division operations and 25 addition/subtractions, whereas the literal method presented in the article uses 28 multiplication/divisions and 18 addition/subtractions. Your improved literal method uses 15 multiplication/divisions and 11 addition/subtractions. If a curve is drawn with the same number of line segments, regardless of the method, then you should expect that the deCasteljau and your improved literal method would calcu-

(continued on page 14)

```
draw_bezier1(bcurve)
BEZIER-BOX *bcurve;
{
 float x, y;
 float tm;
 float t, t2, t3, a, b, c;
 move to((int)bcurve->a.x,(int)bcurve->a.y);

 for (t=0.0; t<=1.0; t += 0.02) {
  t2 = t * t;
  t3 = t2 * t;
  a = 1 - 3*t + 3*t2 - t3;
  b = 3*(t - 2*t2 + t3);
  c = 3*(t2 - t3);
  x = a * bcurve->a.x
      + b * bcurve->b.x
      + c * bcurve->c.x
      + t3 * bcurve->d.x;
  y = a * bcurve->a.y
      + b * bcurve->b.y
      + c * bcurve->c.y
      + t3 * bcurve->d.y;
  lineto((int)x, (int)y);
 }
}
```

Example 2

(continued from page 12)
late in approximately the same amount of time, while the original literal method is still the slowest puppy in the litter.

Encapsulating Memory Allocation

Dear *DDJ*,

The article "Encapsulating C Memory Allocation" by Jim Schimandle (*DDJ*, August 1990) describes a code construct which leads to "memory leakage," where a failure in a sequence of memory allocations results in losing those pieces successfully allocated (see Example 3). I agree that memory leakage is a real problem area; its effect is usually only made noticeable after the passage of time and is not made obvious in one's program by simply testing large cases.

The brevity of constructs such as in Example 3 is quite desirable. In an allocation sequence, explicitly checking success and freeing unusable memory areas after each step may severely contort the code. What seems to be called for is a routine which will correctly clean up after an aborted allocation sequence. I devised a routine, *vfree( )*, which takes a (NULL-terminated) list of pointers to allocated regions and frees them (Listing One). This allows program constructs such as that in Example 4.

The operation of *vfree( )* is quite simple — free each non-NULL pointer in the list. Argument pointers are presented in the same order that allocations were attempted, so all pointers to valid regions are contiguous in the beginning of the list, and no pointers past the first NULL need be considered. When allocating a structure and its fields, the fields must be initialized to NULL (through *calloc( )* or a custom allocator). The last attempt's result does not need to be included in the arguments to *vfree( )*; if it succeeds, the call to *vfree( )* will not be made. If all attempts before it succeed and it is the only one to fail, there is no need to free its space.

This function is not only useful in allocation-error memory recovery. Using *vfree( )* makes the coding of multiple successive *free( )* operations cleaner. In addition, there may be a slight runtime benefit as well: Whenever you must call *free( )* several times in succession, the compiler must generate code to push each pointer on the stack and then code to call *free( )*. Using *vfree( )*, all of the argument-pushing is retained, but only one function call is made.

The idea is quite expandable. Freeing of nested structures can be accomplished by creating a deallocation function for each level of structure, and extending the concept of *vfree( )* to take deallocator/region pairs as arguments. If you need to free a grab bag of pointers, some of which may be NULL (such as in the cleanup stage of a function that could have failed or been cancelled at several places), a form of *vfree( )* can be written which takes the number of pointers as its first argument; it would simply count through the argument list and skip the NULLs.

If the allocation for *fooptr->string* fails, *fooptr* is freed. If the allocation for *fooptr->field2* fails, *fooptr* and *fooptr->string* will both be freed. If the allocation for *fooptr->field3* fails, *fooptr, fooptr->string,* and *fooptr->foo2* will all be freed.

George Spofford
Northampton, Massachusetts

Jim responds: The vfree( ) *routine proposed by George is a good example of using a single routine to handle a failure. However, I do not agree that the brevity of construct in his Example 3 is desired. It is only desired if* malloc( )/free( ) *is your only concern. Most data structures require more cleanup than a simple* free( ). *Often there are files to close, other data structures that must be updated, and state variables that need to change. For the example presented,* vfree( ) *is a perfectly good solution. However, it is not generally applicable. What you really need is a constructor/destructor facility as is found in C++.*

The only code benefit to be derived is in code size. vfree( ) *actually imposes a longer execution time overhead because all the pointers must be copied on the stack twice: once for the call to* vfree( ) *and once for the call to* free( ). *Also, the copy loop in the* vfree( ) *routine takes time. This is not an issue if the failure of the* malloc( ) *call is relatively infrequent.*

DDJ

We welcome your comments (and suggestions). Mail your letters (include disk if your letter is lengthy or contains code) to DDJ, *501 Galveston Dr., Redwood City, CA 94063, or send them electronically to CompuServe 76704,50 or via MCI Mail, c/o* DDJ. *Please include your name, city, and state. We reserve the right to edit letters.*

```
      if (    (fooptr = (FOO *) malloc (sizeof (FOO))) == NULL
           || (fooptr->string = strdup (name)) == NULL) {
        return (NULL);

If the strdup() call fails, the memory allocated for the structure
is lost.
```

Example 3

```
#include <stdarg.h>     /* for variable arg macro def's */

/*---  VFREE  -------  Author:  George Spofford --------------*/
void vfree (void *first,...)
{
     static void    *ptrs[MAX_VFREE];   /* reordering area */
            int      i;
      void    *p;
            va_list  argMarker;       /* variable-args marker */
    if (first == NULL)                /* ensure something to do */
          return;
/***  First, accumulate all pointers into array */
    ptrs[0] = first;
    va_start (argMarker, first);
    for (i = 1, p = va_arg(argMarker, void *);  p != NULL && i <
MAX_VFREE;
                        ++i, p = va_arg(argMarker, void *) )
          ptrs[i] = p;
/***  Now, free them all, in reverse order */
    while (--i >= 0)
          free (ptrs[i]);
}
```

Listing One

```
      if (    (fooptr = (FOO *) calloc (1, sizeof (FOO))) == NULL
           || (fooptr->string = strdup (name)) == NULL
           || (fooptr->field2 = (F1 *) malloc (sizeof (F1))) == NULL
           || (fooptr->field3 = (F2 *) malloc (sizeof (F2))) == NULL
) {
        vfree (fooptr, fooptr->string, fooptr->field2, NULL);
        return (NULL);
      }
```

Example 4

Roll Your Own Object-Oriented Language

Striking a chord with Object Prolog

Michael Floyd

A special relationship exists between developing software and writing music. Among other things, the two disciplines share precise languages containing symbols that, unlike English, carry very specific meanings. Yet, these symbols can be combined in countless ways to orchestrate something that can only be termed "harmonious." More often than not, the unexpected becomes the norm, and more to the point of this article, what could be more unexpected than object-oriented extensions to the Prolog language? The extensions, which I call "Object Prolog," involve an inheritance mechanism and message-passing facility that, when included in a Prolog program, allow you to add objects to your programs. I'll also provide a preprocessor that allows you to define your own object definition language and spits out Object Prolog code.

"But," you say, "I'm no logic programmer." Well, I'm no Brahms either, but you should know that Prolog's unique abilities allow it to serve as a sort of "executable specification," meaning that you can easily rewrite the code presented here in your favorite language. In addition, the parser presented in this article provides a code generator that can be modified to output source in your favorite language.

I should mention that I'm using PDC Prolog (formerly Turbo Prolog) from the Prolog Development Center. The code generated by the preprocessor, however, should run under any Prolog compiler with only slight modifications.

Object Prolog

Object Prolog is a small language set that I've defined to wrap around your code. When written correctly, that code will exhibit inheritance, encapsulation, abstraction, polymorphism, and persistence. Object Prolog treats everything in the system as an object, and, as such, there are no formal classes. As defined here, only single inheritance is supported, although support for multiple inheritance can be added.

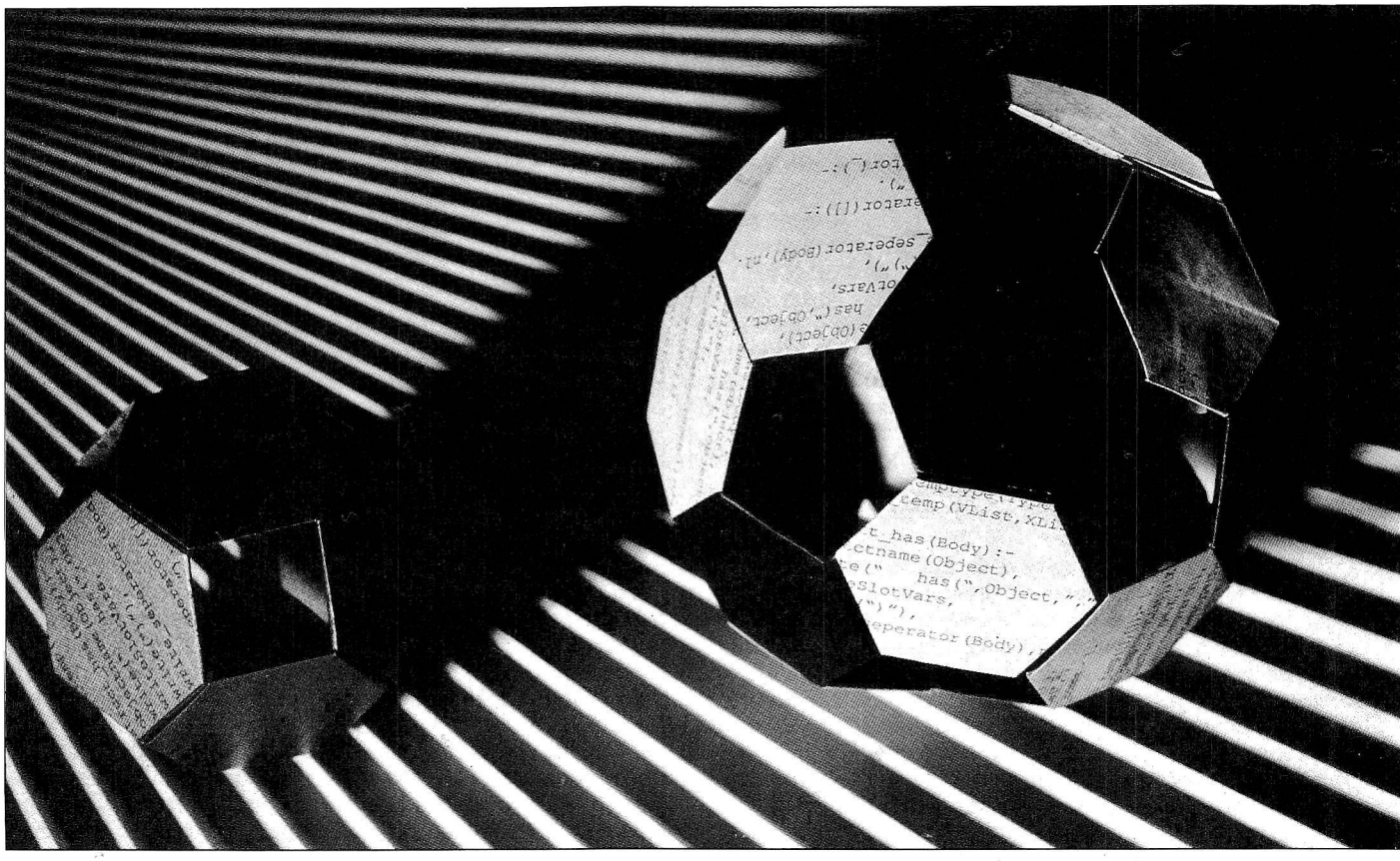
Object Prolog uses four predefined predicates (equivalent to a Pascal procedure) which are defined in OOP.PRO (see Listing One, page 102). The *method( )* predicate is used to define methods in the program and takes an object type and the method name as its two arguments. There is nothing unusual about *method( )* except that it is invoked by the inheritance mechanism. A *method( )*, like any Prolog predicate may call any other predicate in the system, including other methods.

The *msg( )* predicate is used to send messages to objects by taking an object identifier and a message identifier as its arguments and passing the message through the inheritance mechanism to invoke the appropriate method.

The *is_a( )* predicate is a database predicate that simulates a table defining the parent/child relationships in the system. *is_a( )* is generally used only by the inheritance mechanism, although it can be called directly anywhere in the program. One common use for *is_a( )* outside of the inheritance mechanism is to look up the hierarchy for a specific attribute of the parent object. You'll see this action in the discussion of Listing Two.

Finally, the *has( )* predicate is a data structure that represents the dynamic portion of the objects in your program. It has the effect of an instance variable and stores the data associated with an object. *has( )* is a database predicate so that instances of objects can be passed around in memory. The structure for *has( )* consists of an object identifier and a list (similar to a linked-list) of slots which describe the named object. Slots are recursively defined and consist of an identifier and a value. For instance, a slot could be a variable name and the contents of that variable. A value, however, can also contain lists made up from the basic data types, another slot, or a list of slots.

Mike is a technical editor at DDJ *and can be reached directly at* DDJ, *on CompuServe at 76703,4057, or on MCI MAIL as MFLOYD.*



Striking Up the Band

If you are a Turbo Pascal or Turbo C++ programmer, you're familiar with Borland's "Figures" example which is used in the Borland documentation to get budding object-oriented programmers up to speed. The example defines a base object type (or class) called *point* that is used to derive a circle, and later an arc. I'll use that same example, both as a litmus test for Object Prolog and as a common ground for understanding.

Listing Two (page 102) presents FIGURES.PRO, which defines four shapes: point, circle, rectangle, and solid rectangle. The *Clauses* section (similar to procedure definitions) first defines the methods for those shapes. Because PDC Prolog supports the Borland Graphics Interface (BGI) library, I'll use the BGI to draw the various shapes on the screen. The *hide* and *show* methods for *point*, for instance, use BGI's *putpixel( )* routine to display or hide a point on the screen. BGI.PRO (Listing Three, page 104) provides the BGI initialization support.

point is the base object from which all other shapes will inherit. As such, *point* demonstrates the use of abstraction in Object Prolog. In other words, *point*'s purpose is to act as a template for all other shapes that inherit from it rather than to draw points on the screen (although there would be no problem in doing so). *point*'s *init* and *done* methods represent a constructor and destructor, respectively, using *assert* and *retract* to create and destroy objects in memory.

I mentioned earlier that *is_a( )* could be called directly in an Object Prolog program. *method(circle, drag)*, for instance, uses *is_a( )* to send a message to *method(point,drag)*. Note that there is no explicit *circle.drag* method in the Turbo Pascal or Turbo C++ counterparts to Figures. These bindings are handled by a Virtual Method Table (VMT). Object Prolog, however, does not setup a VMT so the lookup is done explicitly by *method(circle.drag)*. This is a dynamic lookup, so nothing is lost in the way of polymorphism, although it does require a little more work.

The Figures example winds up with *arc*, a new object type. *arc* is defined as a child of *point* and therefore reaps the same benefits as *circle*. Note that *arc* is not a stand-alone object file; rather, it is included at the end of Listing Two. The reason is that PDC Prolog requires that all predicates be grouped together. Therefore, *arc* must consult the database and assert the appropriate *is_a* and *has* clauses. There will be more to say about this in the "Encore" section.

A Bow from the Director

One of the beauties of Prolog is its ability to quickly model an algorithm. This is apparent in the instrument that orchestrates the symphony — the inheritance mechanism. (The code for the inheritance mechanism in Listing One, for example, is a paltry 11 lines long.)

The *inherit( )* rules are actually quite powerful and can act in a number of ways. *inherit* can take an object identifier and return its associated value. Given the Figures example, for instance, if *Object* is instantiated to *point*, the first *inherit* clause will call *description* to look for a matching *has* fact in the database. In this case, *description* will return a list of slots containing the values for the *x* and *y* coordinates of *point*.

In the case of *circle*, however, no explicit fact exists. So, a call to *inherit* with *Object* instantiated to circle will invoke the second *inherit* clause, which uses *is_a* to look up the hierarchy until a fact is found. In this case, *inherit* will return the *x* and *y* coordinates with no explicit *has* reference to *circle*.

The second *description* clause comes into play when a *has* clause cannot be found. The assumption is that if it's not an instance variable, it must be a method. If you're concerned about performance here, don't worry. Prolog (like all languages) looks at the data structure before attempting to bind variables, so the instance variables are quickly ruled out.

A point for C programmers to note here is that you can call PDC Prolog from C (both Turbo and Microsoft). You can then get a cheap inheritance mechanism by linking OOP.PRO with your C code and calling *inherit* directly!

Objects in C Minor

Object structures, as represented by *has( )*, can be quite complex, especially if an object slot contains lists of slots within slots, and so on. I decided to build a simple preprocessor that allows you to design objects at a higher level, using what I call an "object definition language" (ODL). The preprocessor defines a syntax similar to Turbo Pascal's, parses the code, and generates Object Prolog source code.

FIGURES.OOP (see Listing Four, page 104) presents an example of the ODL that I've defined. You'll note that this program looks more like the Figures example in the *Turbo Pascal OOP Guide* than Listing Two, with the exception that I've borrowed the style for variable declarations from C. (Note, for instance, the *int* declaration at the beginning of *point.*) Of course, there is no *virtual* keyword for the reasons discussed above.

Turbo Pascal has a nifty style for describing objects. In particular, I like the syntactic style used in defining inheritance relationships: The ancestor is named in the argument to *object( )*. Introducing an *inherit* keyword as defined in "true" Object Pascal works, but is clumsy, and in my opinion, less eloquent. Otherwise, the program is straightforward: Methods are encapsulated directly in the object and support predicates are contained after the object definitions. As with Object Pascal, the *end* statement is used to terminate the object definition and must be followed by a period.

Parsing the ODL

PARSER.PRO in Listing Five (page 105) presents the ODL parser; a top-down parser that operates in a recursive-descent fashion. Because the parser need generate only Object Prolog code, construction of a state machine is not necessary. In fact, much of the defined syntax is a reordering of the Object Prolog syntax so that encapsulation is more natural. Of course, the parser must also create data structures for objects, insert inheritance relationships, and bind static variables.

The algorithm for the parser is implemented in the *scan* predicate. As with many parsers, *scan* does not actually construct a parse tree, but mirrors the tree through predicate calls. The parser takes its input, one line at a time, and passes the line to the lexical analyzer via *tokl* for tokenization. *scan* first looks for the object definition, and then for any variable declarations. The object definition is stored in the database via *insert_isa* for later insertion into the *is_a* table, and the object identifier is recorded in *objectname. Init_vars* then scans the variable string and asserts their tokenized counterparts into the database as slots. Later, in the code generation phase, these slots will be referenced to set up the appropriate *has* clauses.

scan_methods then scans the method identifier and asserts the tokenized method into the database for reference by the code generator. The code generator inserts the object name to create the method clause. Also note that methods may span several lines so *GetMethEnd* is used to search for the end of a method, which is designated by a period (as are traditional Prolog clauses). *scan_methods* continues recursively scanning methods until an *end* statement is reached

Finally, the code generation phase takes over. Code is generated for one object at a time; then *scan* is called again to parse more objects. *generate_code* begins the code generation phase by grabbing the current object identifier from the database and calling *generate_ancestors* to setup the *is_a* table. Next, *generate_methods* is called to insert the object identifier and write any methods stored in memory. *generate_vars* then grabs individual slots and reasserts them as a list of slots, stored as *vars*. Finally, *generate_code* retrieves the *vars* from the database and writes out the *has* clauses.

Because PDC Prolog requires that *like* clauses be grouped together, *has*, *is_a*, and *method* clauses are stored in separate temporary files and merged at the end of the code generation phase. It probably makes sense to store *has* and *is_a* clauses in a separate consult file. In fact, this is a prerequisite for dynamic objects and is required to support object persistence.

As mentioned earlier, the lexical analyzer shown in Listing Six (page 108) is called by the parser via *tokl* to scan and tokenize lexemes. *tokl* uses the built-in *fronttoken* predicate to strip the first lexeme from the input stream, then calls *scan_next* to scan the rest of the stream. *scan_next* also uses *fronttoken* to assert "lookahead" lexemes in the database in the form of a stack. *tokl* next calls *tokenize* to tokenize the string. *tokenize* pulls a lexeme from the input stream and passes the lexeme to *str_tok*, which takes a string and returns its tokenized representation. In most cases, this is a simple lookup. For instance, *str_tok("(", X)* will return the *lpar* token in *X*. But a lookahead is necessary in the case of variable declarations, so *str_tok("int", X)* must pop lexemes off the lookahead stack to get to the variable identifiers. These identifiers are stored in the database as they are popped off the stack.

One other notable point is that *str_tok* suffixes an underscore character to lexemes that coincide with built-in PDC Prolog keywords. For instance, *str_tok* will return *if_* for the *if* lexeme. This is done to resolve conflicts between the object definition language and PDC Prolog language elements.

An Encore

The parser makes a good stepping stone for designing and developing Object Prolog code, but is by no means complete or bulletproof. Indeed, the parser is bare bones at the moment and is crying for improvements. So, if you plan to use the extended language and preprocessor to develop Object Prolog code, you'll probably want to make some additions to the parser and scanner. The first step is to add some error handlers to both the parser and lexical analyzer. And you'll want to add support for any built-in predicates that you need.

As mentioned earlier on in this article, the Figure example included *arc*, rather than linking it as a separate object file. Not only is this a deterrent to polymorphism, but it restricts the size of your object-oriented system. Another problem lies in the fact that every clause in the system (except support clauses) is a *method( )*. Although I'm not certain of the limit, there is a maximum number of clauses allowed for a given predicate. Undoubtedly, someone is going to hit that limit at some point. This is where a method table would come in handy.

My next project will be to build a hierarchy browser to scan the *is_a* relationships and present a tree diagram of the hierarchy. The Prolog Development Center provides window and menu tools, and includes a tree menu predicate that can be cannibalized (most of the toolbox, including tree menu, comes with full source code) to aid in this effort.

In any case, I invite you to pick up the baton and have a go at it. Who knows what melody lurks in the wings, waiting to be played.

DDJ

(Listings begin on page 102.)

Vote for your favorite feature/article.
Circle Reader Service **No. 1.**

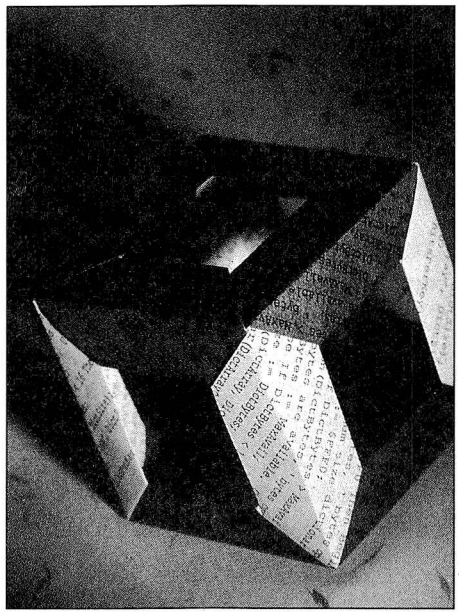
An Existential Dictionary

Superimposed coding packs a lot of information into a small space

Edwin T. Floyd

Have you ever needed a search routine that could determine with fair accuracy whether or not a particular piece of information exists without the overhead of a conventional search? If so, the "superimposed coding" technique I present in this article may be the tool you've been waiting for. I've implemented the system as a Turbo Pascal object that supports superimposed code dictionaries, which I call "existential dictionaries" because they record the fact that a key exists without storing the key itself. Existential dictionaries can be used for a variety of applications, including spell checking, document retrieval, and database applications.

The technique of superimposed coding packs a lot of information into a small space. An existential dictionary for a 10,000-word spelling checker, for instance, can occupy as little as 23K. In the case of database applications, an existential dictionary can save time. For example, consider a 10,000-record database indexed on one field. A superimposed code dictionary for 10,000 keys can be as small as 18K — small enough to hold in memory. Checking the dictionary for a key before searching the index for that key reduces unnecessary index searches, thus reducing such searches by a factor of one thousand.



Small existential dictionaries can be used as "surrogate keys" to help locate documents indexed by author and subject keyword terms. The size of a surrogate key file is typically 10 to 20 percent of the size of a conventional index, and the surrogate key file can be substantially quicker when searching for certain types of queries. Surrogate key files are also used in place of conventional indices in some very large databases.

Edwin is manager of a data center for the Hughston Foundation, a non-profit orthopaedic and sports medicine facility in Columbus, Georgia. He is an occasional contributor to DDJ. *Edwin can be reached through CompuServe at [76067,747].*

How Superimposed Coding Works

Consider a hypothetical tiny spelling checker with a word list of 400 English words. To construct an existential dictionary for it, start with an array of 8000 bits, all initially *off*. For each word in the list, generate a hash number between 0 and 7999, and turn the corresponding bit in the array *on*. If the hash function is sufficiently random (a nontrivial problem), it will distribute the *on* bits uniformly in the table with very few collisions. (A "collision" happens when two words hash to the same bit position.) The code bits for all words are superimposed in the same table, without regard to collisions.

To test an unknown word for membership in the list, first generate the word's hash number. If the corresponding bit is *off*, we are certain that the word is not in the list. If the bit is *on*, the word is either in the list, or its hash number simply happens to match the hash number of a word in the list. This is called a "false drop." (A false drop happens when the existence test succeeds incorrectly due to a collision.) For this dictionary, the probability of a false drop is about 400/8000, or 1/20. We want to reduce this probability as much as is practical.

One way to reduce the false drop probability is to increase the size of the bit table. For instance, we can double the size of the bit table to 16,000 and change the range of the hash function correspondingly. The new probability will then be about 1/40. Jon Bentley, in his priceless book *Programming Pearls*, describes a spelling-checker program written by Doug McIlroy. Doug's program uses a 2^{27} (134 million) bit table to store existence bits for about 30,000 words. His false drop probabil-

(continued from page 20)
ity is a little less than 1/4000. By an ingenious representation trick, Doug manages to compress this 16-Mbyte table to 40K, which, plus an index, makes for a total dictionary size of less than 64K.

Another way to reduce the false drop probability is to represent each word by more than one bit. In our example, we can use two hash functions on each word to turn about 800 bits *on* out of a total of 8000. In this case, the probability for a single-bit collision increases to about 1/10. A false drop now requires two collisions with a combined probability of 1/10 * 1/10 or about 1/100. Thus, using two bits per word improves the accuracy of this dictionary fivefold. Using three bits per word will improve the accuracy even further.

Our dictionary is not limited to Eng-

Derivation of the *BitsOn* Equation

Consider an array of *N* bits, all initially *off*, and an iterative process that selects bits at random and turns them *on*. What is the probability of a "collision," that is, the probability that the random process will select a bit that is already *on*, at the ith iteration? On the first iteration ($i = 0$), the probability is zero because all bits are *off*. On the second iteration ($i = 1$), the probability is *1/N*, because exactly one bit is *on*. Let J_i be the number of bits *on* at the ith iteration. Then, the probability of a collision (call it P_i) is as shown in Figure 4(a). The probability of a non-collision (call it Q_i) is shown in Figure 4(b).

We can see that each time the process succeeds in turning a bit *on* that was previously *off*, the probability of a non-collision is reduced by *1/N*. But the probability of selecting an *off* bit, and thus succeeding in turning it *on*, is the current probability of a non-collision. Combining these facts, we can write a recurrence formula for the value of *Q* at any iteration, based on the previous iteration, see Figure 4(c).

Knowing that $Q_0 = 1$, we can convert the recurrence formula to an explicit equation for Q_i (refer to Figure 4[d]). By substituting and taking the logarithm of both sides, we arrive at the equation shown in Figure 4(e). We can rearrange this equation into the equation shown in Figure 4(f). As *N* becomes larger, the expression shown in Figure 4(g) rapidly becomes so close to -1 that we can neglect the difference. (Hint, it's a "sick" limit.) So, for practical purposes, the equation becomes that shown in Figure 4(h). Solving for J_i, we have the equation in Figure 4(i), which is the expression for the number of bits that were originally *off*, but are now *on*, in a table of *N* bits, after *i* iterations of the random process. In our application, the number of iterations is simply the product of the number of keys and the number of bits per key. So, $i = BK$, and we have the *BitsOn* equation.

The false drop probability is P_i^B. By differentiation, we can show that $J_i/N = 1/2$ minimizes the false drop probability. (The proof is left as an exercise to the reader. Hint, differentiate $[BitsOn/N]^B$ rsp. *B*.) So, in an optimal table, the relationship shown in Figure 4(j) holds true. Solving for *N*, we arrive at the optimal table-size equation shown in Figure 4(k). This is the expression for the size in bits of an optimal bit table containing *K* keys with *B* bits per key.

The value of $-Ln(0.5)$ showed up early in the process of developing these routines. I experimented with various sizes of bit tables by counting iterations while turning on random bits, until half of the bits were *on*. The ratio of iterations to table size in bits was always about 0.693. This derivation was motivated by a desire to explain this mysterious number.

— E.F.

(a) $$P_i = \frac{J_i}{N}$$

(b) $$Q_i = 1 - P_i = 1 - \frac{J_i}{N} = \frac{N - J_i}{N}$$

(c) $$Q_i = Q_{i-1} - Q_{i-1}\frac{1}{N} = Q_{i-1}\frac{N-1}{N}$$

(d) $$Q_i = \left(\frac{N-1}{N}\right)^i$$

(e) $$\mathrm{Log}_e(Q_i) = \mathrm{Log}_e\frac{N - J_i}{N} = i\mathrm{Log}_e\frac{N-1}{N}$$

(f) $$\frac{i}{N} = \frac{\mathrm{Log}_e\frac{N - J_i}{N}}{N\,\mathrm{Log}_e\frac{N-1}{N}}$$

(g) $$N\,\mathrm{Log}_e\frac{N-1}{N} \to -1$$

(h) $$\frac{i}{N} = -\mathrm{Log}_e\frac{N - J_i}{N}$$

(i) $$J_i = N - Ne^{-i/N}$$

(j) $$\frac{J_i}{N} = \frac{N - Ne^{-i/N}}{N} = 1 - e^{-i/N} = \frac{1}{2}$$

$$e^{-i/N} = 1 - 1/2 = 0.5$$

(k) $$N = \frac{i}{-\mathrm{Log}_e(0.5)} = \frac{BK}{-\mathrm{Log}_e(0.5)}$$

Figure 4: BitsOn *equations (a): Collision probability (b): Non-collision probability (c): Recurrence formula (d): Explicit equation (e): Logarithm (f): Rearranged (g): Limit (h): Approximation (i):* BitsOn *(j): Optimal table (k): Solved for* N.

(continued from page 22)
lish words. In general, any sequence of bytes, or "key" can be represented as a set of bits in a bit table. The number of bits used to represent a key is independent of the length of the key. As more bits are used to represent each key and the bit table fills up, two effects become noticeable.

First, more collisions occur. The number of *on* bits in the table is significantly less than the number of keys multiplied by the number of bits per key. In fact, if our hash functions randomize well and the bit table is not too small, the number of *on* bits will be very close to the value predicted by the *BitsOn* equation in Figure 1(a). In this equation, N represents the size of the table in bits, e is the natural logarithm base, K represents the number of keys, and B represents the number of bits per key. (See the accompanying textbox entitled "Derivation of the *BitsOn* Equation.") If we use ten bits per word in our hypothetical dictionary, then N= 8000, K= 400, and B=10. According to this formula, about 3150 bits are *on*.

In general, the false drop probability is the product of all the single bit collision probabilities, each of which is *BitsOn/N*, see Figure 1(b). This equation yields a false drop probability of $(3150/8000)^{10}$, or about 1/11,000, as shown in the example.

The second effect of using more bits to represent each key is that the accuracy of the dictionary continues to improve more and more slowly until half of the bits in the table are *on*. At that point, the accuracy begins to deteriorate. At the optimal point, when half of the bits are *on*, the false drop probability is $1/2^B$. Given the number of keys and the number of bits per key, we can rearrange the equations described earlier into a simple formula for the optimal table size, as shown in Figure 1(c).

(a) $\text{BitsOn} = N - N\,e^{-BK/N}$

(b) $\text{FalseDrop} = \left(\frac{\text{BitsOn}}{N}\right)^{B}$

(c) $N = \frac{KB}{-\text{Log}_e(0.5)} \approx \frac{KB}{0.693147..}$

(d) $\text{Expected collisions} = K - N + N\,e^{-K/N}$

Figure 1: *Dictionary equations (a):* BitsOn *equation (b):* FalseDrop *equation (c): Optimal table-size equation (d): Expected collisions*

Small existential dictionaries can be used as "surrogate keys" to help locate documents indexed by author and subject keyword terms

To use Doug McIlroy's problem as an example, we have 30,000 keys and we want the false drop probability to equal 1/4000 or less. Representing each key by 12 bits in an optimal bit array will yield a false drop probability of $1/2^{12}$, or 1/4096. In this case, the array size will be 30,000*12/0.693,147 = 519,370 bits, or 64,921 bytes, which is somewhat larger than Doug's dictionary. The procedure for building our bit table is considerably faster and simpler than Doug's procedure, which requires a *sort*. Also, by making our bit table a little larger than the optimal size, we can add keys dynamically. For instance, a 65,520-byte table at 12 bits per key becomes optimal at about 30,277 keys. At 31,000 keys, the false drop probability is still only 1/3369.

The Hash Function

With these formulae, we can engineer a bit table for any number of keys and with any desired degree of accuracy. The calculated performance is guaranteed, as long as the hash function is sufficiently random. I evaluated a number of hash functions found in the literature for speed and randomization and eventually created one based upon a random-number generator described by Steven Park and Keith Miller in a 1988 *Communications of the ACM* article entitled "Random Number Generators: Good Ones are Hard to Find." The "Minimal Standard" generator described in their article is a simple multiplicative congruence algorithm that generates a sequence of seeds:

Next seed = (seed*16807) modulo 2,147,483,647

The Park-Miller generator is a 31-bit, full-period generator. Given any 31-bit starting seed (except zero), it will not return to that seed or repeat a number for 2,147,483,647 cycles. The sequence passes most tests of randomness. It is not, of course, truly random. A given starting seed always generates the same sequence of "random" numbers. Because of this, we call it a "pseudo-random" number generator.

In the January, 1990 *Communications of the ACM*, David Carta described a fast machine-language implementation of the Park-Miller generator. My version of the Carta implementation for the 8086 instruction set requires only two 16-bit multiply operations, plus some shifts and adds.

A good pseudo-random number generator simplifies the task of creating hash functions. The generator eliminates the need for a separate hash function for each dictionary bit. Instead, we need only a single hash function to produce a suitable starting seed. The random number generator will then beget an endless supply of unique, random, 31-bit integers. Each integer, modulo N, is a bit position to turn *on* or *test*.

We still need to construct a suitable hash function to generate the starting seed. I found this task unexpectedly frustrating. The entire responsibility for avoiding collisions rests with the seed-generator function. If this function generates the same seed number for two different keys, the bits selected will also be the same for those two keys. Therefore, the seed function must generate as few 31-bit seed collisions as possible among the keys of interest and should also be quick. What is a reasonable collision performance to expect from a hash function? I set out to devise a suitable test.

First, I needed a large number of typical keys. I started with a list of 109,584 unique English words obtained from a public domain source. The words averaged 8.5 characters each. I planned to use a hash function to generate a 31-bit number for each word. If the hash function randomized well, it would produce about the same number of collisions as a random selection of 109,584 items from a population of 2^{31}, with replacement. I computed the number of collisions to expect in a random selection of K items from a population of N, see Figure 1(d). Ac-

(continued on page 28)

(continued from page 24)
cording to this equation, for K=109584 and N=2147483647, I could expect about three collisions, which is not enough for a meaningful test.

Accordingly, I expanded the number of keys. I generated three keys from each English word: All uppercase, all lowercase, and first letter capitalized with the rest of the word lowercase. (I excluded the single-letter words "A" and "I.") I now had 328,751 keys. I also included the 10-digit ASCII numbers from zero to 109,583, which brought the count to K= 438335 unique keys and 45 expected collisions. To evaluate each hash function, I used it on the keys to generate 438,335 31-bit numbers. Next, I sorted the numbers and counted the duplicates. Throughout the remainder of this article, I'll refer to this procecure as the "collision test."

In addition, I devised a realistic test of the hash function's false-drop performance. I partitioned the word list into seven disjoint subsets of 8000 to 18,000 words each. To evaluate each hash function, I used it to create an optimal bit table for each subset, using 14 bits per word. I shifted each word to lowercase before hashing. Then, for each subset, I counted the number of false drops from the entire word list. I left all the test words in uppercase, so that none of them matched any word in a subset. The false drop probability for each bit table is about 1/16,384. So, a good hash function should produce about 7*109,584/16,384, or about 47 false drops for all subsets combined. In the following text, I'll refer to this procedure as the "practical test."

Existential dictionaries can be used for a variety of applications, including spell checking, document retrieval, and database applications

The behavior of hash functions on this scale is somewhat counterintuitive. A widely used, 32-bit CRC function, for instance, generated thousands of collisions. Eventually, I ran across the following reference to a simple hash algorithm in Donald Knuth's *The Art of Computer Programming*:

1. Generate a sequence of as many pseudo-random numbers as there are bytes in the key.
2. Multiply each byte in the key by a different member of the sequence.
3. Add up the results.

Since I had a source of random numbers, I tried this hash algorithm with an initial seed of "1." The resulting collision count was 61. This result was close, but there were still two problems with this algorithm. It ignored trailing null bytes, and, unless the starting seed was varied, it used the same sequence of multipliers for every key. The first problem was easy to fix — I simply added a constant to each byte before multiplying. (I used 65,280 *[FF00h].*)

I am ashamed to say how many blind alleys I ran down before settling on a solution to the second problem. Because the starting seed completely determines the random number sequence, you might think that enough variation could be easily introduced by setting

(continued from page 28)
the seed to a simple function of the length or the terminal key characters. Not so. The only reliable method I've found to randomize the starting seed is to run the hash function twice in the following manner: The first time you run the function, start with a constant seed. The second time around, use the result of the first run as the seed.

Each different starting seed defines a different hash function. I tested 50 starting seeds and discovered that most of them performed well on the collision test, but poorly on the practical test. I knew that after the hash function determines the initial seed, the practical test uses the raw random-number generator to select bits. This fact eventually led me to suspect the "randomness" of the Park-Miller generator's output. Knuth describes a simple shuffling algorithm that improves the randomness of even poor generators. After I incorporated an eight-way shuffle into the Park-Miller generator, the false drop count improved significantly, although it was still 17 percent above the expected count. The eight-way shuffling algorithm is shown in Figure 2.

```
Establish the initial seed.
Build the shuffle table:
    fill an eight entry table with the next eight Park-Miller numbers; save the ninth
    Park-Miller number in a variable called NextOut.

To generate the next shuffled random number:
    select a table entry using NextOut, bits 17 through 19; replace NextOut with the
    contents of the table entry; replace the table entry with the next Park-Miller number;
    return the contents of NextOut as the random number.
```

Figure 2: *The eight-way shuffling algorithm*

Seed	Coll	False-Drop Raw	False-Drop Shuf	Seed	Coll	False-Drop Raw	False-Drop Shuf
1	36	64	61	16	41	51	62
2	48	66	48	17	37	57	58
3	53	55	49	18	56	72	59
4	41	67	41	19	38	80	65
5	40	59	49	20	47	80	72
6	48	47	62	21	46	67	54
7	47	62	54	22	39	71	44
8	47	64	58	23	50	70	48
9	36	63	64	24	33	57	64
10	43	69	54	25	46	66	56
11	43	59	49	26	35	46	58
12	45	67	51	27	51	59	46
13	39	68	48	28	40	67	55
14	40	65	50	29	51	60	67
15	48	68	47	30	53	54	58
				Average:	44	63	55

Table 1: *Seed performance*

```
Function Hash (seed, key)
    Set the initial value of the function result to zero.
    For each byte in the key:
        generate the next Park-Miller seed;
        add [(key byte + 65,280) * seed] to the function result.
    Limit the final result to the low-order 31 bits;
    if it's zero, increment it.

Employ the function twice:

    Hash (Hash (26, key), key)

The final hash result is the initial seed for generating bit numbers:

    Build the shuffle table.
    Repeat B times:
        generate the next shuffled random number;
        set (or test) the bit specified by the number modulo N.

If you have a fast machine and wish to set more than 26 bits per key:

    Set the initial seed to 1.
    Repeat B times:
        Set (or test) the bit specified by Hash(Hash(seed,key),key); increment the seed.
```

Figure 3: *Recommended hash and bit selection procedure*

Superimposed coding is an elegant way to test for key existence

Even shuffling will not randomize effectively enough where extreme accuracy is required. An optimal bit table for 26 or more bits per key will suffer from the unavoidable collision behavior of the initial hash function. In such a case, I would not use the random-number generator at all. Instead, I would repeat the hash function with a different starting seed for each bit. This method would require a fast processor to achieve good performance.

Table 1 shows the first 30 seeds and their collision and false-drop counts. Repeating the hash algorithm for each bit produced 43 false drops.

There is no evidence of any real pattern here, and no reason to believe that any particular starting seed is better than the others for all possible key sets. I selected a starting seed of 26, which led to the recommended hash- and bit-selection procedure outlined in Figure 3. This procedure produces bit tables that perform according to the design information given earlier.

Implementation

The implementation presented here is a Turbo Pascal unit (see Listing One, page 110) that supplies a *Dictionary* object; the methods are listed in Table 2. The two non-object functions are listed in Table 3. (Refer to the actual code for details about how to use these routines.) I've also developed a complete batch-oriented spelling checker to illustrate the use of the dictionary objects. Due to space constraints, the spelling checker isn't listed in this article, although the complete program (including documentation and sample terms), is available through *DDJ*. (Refer to the "Source Code Availability" section on page 3.)

The save file produced by the *SaveDictionary* method is intended for di-

(continued from page 30)
rect, binary portability to another hardware platform, a Tandem NonStop minicomputer. The only concession made to the Tandem architecture is the byte order of binary integers. These are byte-reversed before saving to disk, and reversed again after restoration.

The random-number generator, the hash function, and the bit-set and test functions are implemented as inline-assembler macros. The bit table size is limited to 65,520 bytes as a concession to Turbo Pascal's maximum *GetMem* size, and to improve the performance of the bit-set and test functions. You can overcome this limitation by distributing keys to multiple dictionaries. For instance, my Tandem spelling checker divides the dictionary by the first letter of a word into seven bit tables: A-B, C-D, E-H, I-N, O-R, S-T, and U-Z. These are the same partitions used in the practical test. Observe one caution when using multiple bit tables: You must add together the false drop probabilities for each existence test performed on the same key. For instance, if you test a key in two dictionaries and each dictionary has a false drop probability of 1/4096, the combined false drop probability is 2/4096 or 1/2048. My spelling checker selects the appropriate dictionary for the unknown word by the word's first character, and tests only in that dictionary.

Performance

To get an idea of the speed of the dictionary methods, I performed a timing benchmark on an 8-MHz NEC-V20. (This machine runs about three times as fast as the original PC.) First, I built an optimal, 14-bit dictionary from a subset of the 109,584 word list. The subset consisted of the 17,639 words beginning with "S" and "T." The false drop probability, as determined by counting the *on* bits, was 1/16,450.

Secondly, I read the entire 109,584 word list and tested each word against the dictionary. Actually, I read each list twice; first I timed the read alone, and then I timed the read with the dictionary method calls. The differences in timings were an estimate of the processor time spent in the dictionary methods. In this benchmark, the methods inserted 250 keys per second and tested 435 keys per second. Existence testing was faster than inserting because almost all of the words, excepting those beginning with "S" and "T" were rejected. The existence test terminates as soon as it finds the first zero bit. Key length and number of bits per key affect performance. For instance, with the same files at 12 bits per key, the benchmark inserted 273 keys per second and tested 447 keys per second. At 14 bits per key, on a 20-MHz 80386, the benchmark inserted 1558 keys per second and tested 2611 keys.

The number of iterations of the Park-Miller algorithm determines the performance of the hash and bit-selection algorithms. The hash algorithm performs $2L$ iterations, where L is the length of the key. The total number of iterations for the hash and bit-selection algorithms is $2L+B+9$, where B is the number of bits per key. If we repeat the hash function for each bit, we perform $2BL$ iterations. When the hash function is repeated for each bit, the 80386 benchmark can insert only 361 keys per second and can test 1712 keys.

Method	Description
Init(MaxKeys, BitsPerKey)	A constructor that calculates the size of the bit table. Acquires and initializes storage from the heap.
InsertString, InsertBlock, and *InsertHash*	Boolean functions which set bits in the bit table. These functions return TRUE if all the bits are already *on*.
StringInDictionary, BlockInDictionary, and *HashInDictionary*	Boolean functions which return TRUE if all bits associated with a key or initial hash and *HashInDictionary* seed are *on* in the bit table.
SaveDictionary and *RestoreDictionary*	Preserve the attributes and contents of the bit table in a disk file.
EstError and *ActError*	Floating-point functions which return the false drop probability. *EstError* estimates it by the *BitsOn* and *FalseDrop* equations; *ActError* counts the *on* bits and uses the *FalseDrop* equation to compute the exact probability. In numerous tests, these functions have always returned very nearly the same result.

Table 2: *Methods for* Dictionary *object*

DictionaryBytes(MaxKeys, BitsPerKey)	A *LongInt* function that uses the Optimal Table Size equation to compute the size in bytes of an optimal bit table for the specified key count and bits per key.
DictHash(Data, Size)	A *LongInt* function that provides access to the hash algorithm.

Table 3: *Non-object functions*

Conclusion

Superimposed coding is an elegant way to test for the existence of a key. This approach has applications in many areas of information retrieval. It is a fast method, especially with the shortcuts afforded by the Park-Miller random-number generator. The equations available allow us to engineer an optimal existential dictionary with almost any desired capacity and accuracy.

Further Reading

1. Bentley, Jon. *Programming Pearls.* Reading, Mass.: Addison-Wesley, 1986.

2. Berra, P. Bruce; S.M. Chung, and N.I. Hachem. "Computer Architecture for a Surrogate File to a Very Large Data/Knowledge Base." *IEEE Computer* (March 1987).

3. Carta, David G. "Two Fast Implementations of the 'Minimal Standard' Random Number Generator." *Communications of the ACM* (January 1990).

4. Knuth, Donald E. *The Art of Computer Programming.* vol. 2, pp. 32–33; vol. 3, pp. 559–567. Reading, Mass.: Addison-Wesley, 1973.

5. Park, Stephen K., K.W. Miller. "Random Number Generators: Good Ones are Hard to Find." *Communications of the ACM* (October 1988).

6. Peterson, James L. "A Note on Undetected Typing Errors." *Communications of the ACM* (July 1986).

7. Salton, Gerard. *Automatic Text Processing.* Reading, Mass.: Addison-Wesley, 1989.

DDJ

(Listing begins on page 110.)

Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

Object-Oriented Debugging

Strategies and tools for debugging your OOP apps

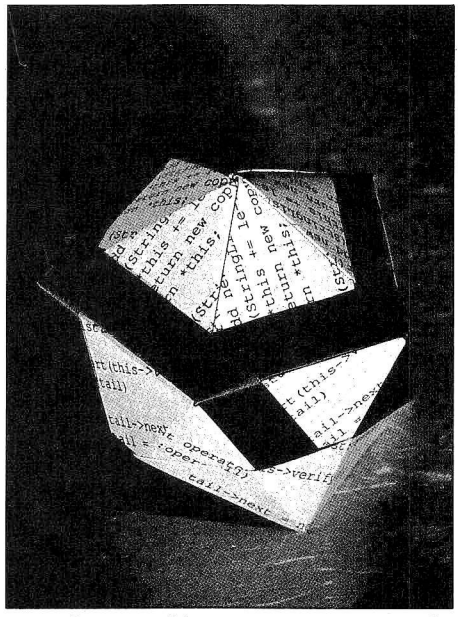
Simon Tooke

This article discusses strategies and tools for object-oriented debugging. Because of its facilities for data abstraction and inheritance, and because of its similarities to C, the language used to demonstrate the principles in this article will be C++, and a good portion of the content will be C++ specific. However, many of the concepts presented here are applicable to other object-oriented environments.

C++ Debugging is Different

Typical C applications tend to be procedure oriented. Execution flows from point A to point B, and somewhere in between, a function may be called to manipulate data, read it in, or write it out. Functions may have side-effects involving many pieces of data or none. When you debug, you watch statement execution to ensure that everything happens in expected order.

In a well-designed C++ application, the "methods" (member functions) available to manipulate an item of data are well defined. Access to data is restricted via a clean set of interface routines. Because of this, the state of the data is very important and is what we are typically concerned with when verifying the correctness of C++ code. Thus, intelligent access to data is as important as program execution flow when searching for a problem. Designing special debugging routines from the beginning can make accessing that data less of a headache for the programmer.



C++ compilers enforce much stricter type checking than traditional C compilers. This all but eliminates several problems such as calling functions with the wrong arguments, type mismatches in assignments, and attempts to store data into constant objects. There are far more original ways to shoot yourself in the foot these days.

A Sample Class

For examples presented in this article, I'll use a simple string class. Several related classes, such as a class containing a list of strings and a class that iterates over a list of strings, are presented. These classes are declared in Listing One (page 114); the *main( )* program is in Listing Two (page 114).

Many of the functions will be trivial when applied against these small classes; it is not my intent in this article to advocate bogging down a 20-line class with several hundred lines of extra debugging code, but to demonstrate practices that are useful when dealing with more complex classes.

The *String* class provides us with a container for simple text strings. Storage allocation for buffers is performed via *malloc( )*, and the internal pointer *s* will be guaranteed to always point at a null-terminated string of length 0 or greater.

A *StringList* is an ordered container for *Strings*. This class is implemented as a singly linked list of *StringListElements*, each of which points to a *String* and to the next element in the list.

By declaring an instance of a *StringListIterator*, a method is provided to examine the individual entries in a *StringList* without having any knowledge of the internals of *StringList*.

Debug From the Start

Too often, debugging is left as the final stage of development, because problems only become evident after a program is compiled and run for the first time. Think about debugging while designing data structures. This forces you to ask questions such as "How can I prove that data is correct?" "How does this function modify the data?" and

Simon is a member of SCO Canada's C++ development tool project and can be contacted at 130 Bloor Street West, 10th Floor, Toronto, Ontario, Canada M5S 1N5.

(continued from page 36)
"What assumptions are being made here?" at a point where something can be done about weaknesses in design.

You can usually afford to be inefficient with debugging code, but you must write carefully; incorrectly flagging an error can lead to hard-to-find problems. At the same time, debugging libraries and user debug code are also prone to errors. I once spent four days tracking a problem in my application because a beta version of a debugging *malloc( )* library incorrectly told me I had wild data pointers — and then had the gall to abort my program.

Using Assert Macros

Long familiar to C users, the *assert( )* macro (which is trivial to write, but is supplied by most compiler vendors) finds a good home in C++. This macro (defined in Listings Three and Four, page 114) takes an expression as a parameter, and, if the expression is false (has the value 0), usually halts the program and prints a message giving the line number and file where the problem occurred. This tendency to abort means that a program with *assert( )* scattered throughout is less resilient, *if* that program is not fully debugged.

At any point where an important assumption is made about the correctness of calculated data, insert an *assert( )* call, which will display a message if that assumption is invalid. *assert( )* macros should not be used to test input data for correctness; that always belongs within the program proper because it is a likely occurrence. If a *switch* statement doesn't have a *default:* case, add one, wrap it in *#ifdef DEBUG*, and add an *assert( )* within it. For portability, never include character strings within the condition; many implementations do not handle this properly.

In Figure 1, *StringListIterator::reset( )* (a function that restarts the iterator instance on the current class) requires a pointer to the *StringList* to be iterated over. The first *assert( )* will ensure that the current instance is valid before we start operating on it. The second *assert( )* ensures that this instance already points to a *StringList* before trying to access the beginning of that list. Although an instance of *StringListIterator* may be created without a list to iterate over, it is an error to perform a *reset( )* or a *next( )* without previously pointing to a *StringList*.

Find invisible compiler-generated temporaries by examining the C output

assert( ) is a macro, and because of this it can vanish when a program is compiled for production use. All code that is added for debugging only should be wrapped in preprocessor *#ifdef DEBUG/#endif* directives so that it vanishes in production programs when *DEBUG* is not defined.

Nevertheless, it may be desirable to always compile *assert( )*s into a program to ensure that problems in the field are properly detected. Yet often performance, speed, and robustness considerations outweigh this desire. A program that keeps running may be more useful to a user than one that aborts because an *assert( )* for a trivial (or worse, nonexistent) error has been triggered.

Verifying Data Structures

When debugging a live program, it is quite easy to display an integer, string, or other simple data structure, look at it, and say "Yes, this looks good." Verification of more complex structures may involve several layers of pointer indirection. It may be tedious to examine all elements in, say, a linked list by hand, following all the pointers through until the end.

If you write functions to check your structures, these can be called within your program and directly by you at debug time. For instances of classes, these should be virtual member functions so that the correct routine is called for instances of derived classes accessed through base class pointers. Call these functions at the start and end of all other member functions to flag corrupted data and assist in isolating the problem.

If an assert fails at the start of the function, some agent external to the class was able to modify the data. If an assert fails at the end of the function, the function itself incorrectly modified the data. These functions are useful both combined with *assert( )* macros and when called from the debugger. An example of *verify( )* for the *StringListIterator* can be found in Listing Five (page 114).

Displaying Data Structures

Examining a complex structure by hand can be tedious in exactly the same way that verifying that structure is. The structure may be complex, involve multiple pointer indirections, or simply be so big that only a small portion is actually useful. It is therefore advisable to add functions to display data structures in some concise fashion. Again, within a class hierarchy these should be virtual member functions. These may never be called by application code, but are useful when called by the debugger. The *dump( )* function that displays data for *StringList* is also found in Listing Five.

Data verification and data display routines should be named consistently. I chose *verify( )* and *dump( )*, so that I always know what to call for any given structure.

Use Debugging Libraries

Since data is often the primary concern of a C++ program, it follows that many problems may surround the functions of memory allocation and deletion. Linking with one of the many readily available debugging *malloc( )* libraries allows runtime checks to ensure that nothing runs past the end of a malloc'd area and that only valid pointers are passed to *free( )*.

In a C++ program, one has the option of overloading *new* and *delete* on a class-by-class basis to gather statistics, check for potential problems, or find "memory leaks." Depending on the compiler implementation, *new* and *delete* often eventually call *malloc( )* and *free( )*, in which case a debugging *malloc( )* library is useful here, too.

Deep and Shallow Copy

Many C++ data structures differentiate between "deep" and "shallow" copies. A deep copy copies the structure itself

```
//          reset( ) - rewind iterator to first element of list
void StringListIterator::reset(void)

{
        // ensure valid data
        assert(verify( ));
        // we cannot reset if no list
        assert(list != NULL);
        // point to beginning of list
        nxt = list->head;
}
```

Figure 1: *Using the* assert( ) *macro*

(continued from page 38)
and any items that pointers in the structure point to. An example of a deep copy is *String::operator=* in the example classes. Deep copies ensure independence between different instances of a class by producing an entirely new copy of a structure. Shallow copies usually copy the structure itself only, but any pointers are left pointing at their original targets.

Shallow copies are quick and good for making a read-only copy of an object. Since the shallow copy does not have control over the items it points to, care must be taken not to destroy those objects during the normal use of either the original or the copy. Often, items pointed to by objects that implement shallow copies contain reference counts, so they are only destroyed when they are no longer required by any object. The Stroustrup book contains a good example of a string class implemented using reference counts. If reference counts are used, the shallow copy is usually as good as a deep copy.

C++ programs tend to be more data-driven than those in C

One typical problem involves destroying the original object after a shallow copy has been made. The pointers within the copy end up pointing at free storage, which may "look" perfectly normal until it is reused, some time later. A related (and usually identical) problem arises if the copy is destroyed; then the original may be invalidated.

```
Thing& operator +(Thing& one, Thing& two)
{
        Thing* p = new Thing;
        // . . . .perform some magic . . .
        return *p;
}
```

Figure 2: *Hidden memory leak*

```
struct baseClass
{
        int i;
        virtual void dump( )
        {
                cerr << "i=" << i << "\n";
        }
};
struct derivedClass : baseClass
{
        int j;
        virtual void dump( )
        {
                // The line below is wrong:
                ((baseClass *)(this))->dump( );

                cerr << "j=" << j <<"\n";
        }
};
```

Figure 3: *Incorrect use of pointer casting*

```
struct Class1 { int i; };
struct Class2 { int i; };

void func(Class1& ref)
{
        ref.i = 5;
}

main( )
{
        Class2 *ptr = new Class2;
        ptr->i = 1;
        cerr << "before, i=" << (ptr->i) << "\n";
        func((Class1)(*ptr));
        cerr << "after, i=" << (ptr->i) << "\n";
}
```

Figure 4: *Invisible (and silent) temporary created by a cast*

Memory Leaks

A common mistake in writing an overloaded operator is to allocate a new instance of a class, calculate the new value of this instance, and return the pointer to this new instance. This will work, but the compiler has no way of knowing that it should free up the new data after it has finished with it. A program may work perfectly well on small amounts of data but crash after it has used most of available memory. This is called a "memory leak." The code fragment in Figure 2 illustrates one such case.

Multitasking systems such as Unix allow you to take a suspect section of code and loop through it again and again while watching the size of the process grow and grow, but DOS does not easily allow this. With DOS, a good debugging *malloc( )* library is (again) useful. Most debugging *malloc( )*s can produce a log of *malloc( )* and *free( )* calls which can then be examined for problems.

Trivial Casting

Casting a pointer from an instance of a derived class to a base class does not change the virtual functions associated with that instance. In particular, the example in Figure 3 will loop forever if *derivedClass::dump( )* is called. Since *baseClass::dump( )* is accessed through the virtual function table (often referred to as the "vtbl" due to one implementation of this feature), and the virtual function entry in an instance of a derived class points to *derivedClass::dump( )*, the statement *((baseClass *)(this))->dump( )* will simply call *derivedClass::dump( )* recursively. In this case, a proper call to the desired function would use the syntax *baseClass::dump( )*.

In Figure 4 (ignoring the fact that this example is contrived and is poor coding practice), the compiler silently creates a temporary variable of type *Class1* and passes a reference to this temporary into *func( )*. When *func( )* returns, the modified temporary is deleted and the value of the real *ptr->i* is unchanged. This all happens silently in most C++ compilers, and is a source not only of inefficiency, but also errors. Looking at the C output (if available) confirms this, but it is a hard problem to catch unless you suspect it exists.

Using a Debugger

Use a good debugger to help to know your program. Be aware of its capabilities and expand them through the ad-

(continued from page 40)
dition of debugging routines where it falls short.

Most C debugging sessions consist of stack traces, setting simple breakpoints, and single-stepping. Since C++ is more data-driven, debugger features such as conditional breakpoints suddenly become very useful. For example, setting a breakpoint in a member function for a particular instance of a class may become trivial if your debugger supports commands of the form "Stop in ClassName::Function if this==&instance."

Handling all of C++ features in an intelligent way is a challenge for a debugger. A good debugger will allow you to think in C++, type in C++ names, and call C++ functions properly. Better yet, it will resemble a Smalltalk browser, automatically cross-referencing classes and instances.

Compiling with *-g* (or the equivalent) adds debugging information to an executable, but there may be other switches on your compiler that provide increased functionality. For example, some debuggers are incapable of setting breakpoints in inline functions unless inlining has been turned off at compile time. Listing Six (page 115) shows the Make file for our sample programs.

If the C++ implementation is capable of it, it may produce C source code as one of the steps in the compilation sequence. There are debuggers that allow you to step through the C code in much the same way as you would step through assembler when debugging C.

Null Pointer Dereferencing

In C++, a common problem is to call a member function using a null pointer to instance. If the pointer *ptr* is NULL, then *ptr->function( )* behaves quite differently, depending on the type of the function. If *function( )* is static, it will be called and nothing out of the ordinary will occur; the pointer value is not passed. If *function( )* is non-virtual, then it will be called, but references within the function will attempt to access data at location 0. If, however, *function( )* is virtual, then C++ will dereference the pointer in an attempt to access the virtual function table for that class. An (effectively) random pointer to function will be read, and the program will probably fly south for the winter.

Debuggers with hardware capabilities (such as add-in boards or access to CPU debug registers) usually allow trapping on read access. This feature is not only useful to trace program access to a particular variable, but, when location 0 is trapped, should catch attempts to access data through null pointers.

Conclusion

The key to C++ (as to most object-oriented languages) is that only a small, known section of the program deals with manipulating a given type of data. If that data is wrong, then only a known portion of the program (the member functions) can be responsible. Careful design will insure that any given member function expects an internally consistent object when called, and leaves the object correct on leaving.

Source Code Availability

As a service to our readers, all source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Drive, Redwood City, CA 94063, or call 800-356-2002 (inside Calif.) or 800-533-4372 (outside Calif.). Specify issue number and disk format. Code is also available through M&T's Telepath online service (via Tymnet) and through the DDJ Forum on CompuServe (type GO DDJ).

DDJ

(Listings begin on page 114.)

Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

CTrace
A Message Logging Class

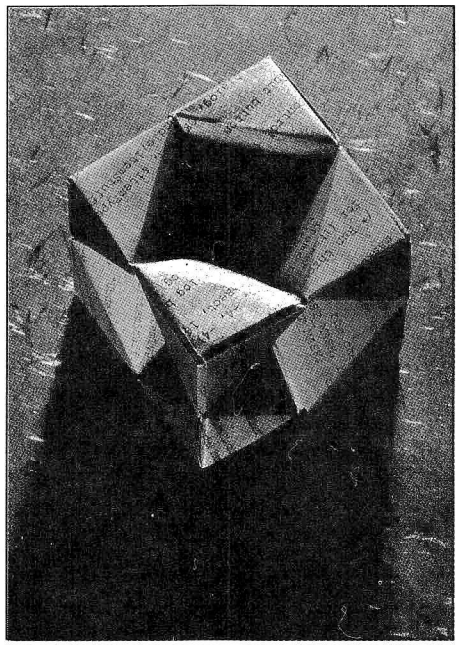
A tool that augments your development environment

William D. Cramer

If you're like me, you've probably had some exposure to more traditional development environments on more traditional platforms. You know the sort of environment I mean — a top-to-bottom program running on a dumb-terminal or PC. The kind of platform where just a few well-placed *printf( )*s can mean the difference between taking two minutes to pinpoint an error and taking two hours.

Most Macintosh programs provide the user with a very visual, very interactive interface. When developing such applications, you might test a particular feature by playing the role of the user — if you see an anomaly, you use the Think C debugger to set a few key breakpoints, and then repeat your steps until you hit one of them. At that point, you can poke around until you find some stray pointer or uninitialized variable and correct the problem.

However, there's a whole set of problems that don't lend themselves to this kind of debugging. For example, actions involving a good number of calculations, iterations, or deeply embedded function calls — all based on a single user action — add complexity if a problem occurs. Then, too, there are operations that rely on certain timing constraints: If the user is expected to respond within a given time period, for example, debugging with breakpoints can ruin the whole timing frame. Finally, programs that deal with a non-human interface (such as AppleTalk or MacTCP) can cause no end of heartache when that external interface doesn't play fair.



This article describes a simple tool that augments your development environment with a general-purpose message logging window. The tool, built out of existing Think Class Library objects, furnishes you with some basic *printf( )* capabilities and lets you define categories of messages. By enabling and disabling these categories, you can selectively control which events get logged.

Bill is a software engineering project manager for IEX Corp., a Plano, Texas engineering and consulting firm. He can be reached via e-mail at uunet!iex!cramer.

What is CTrace?

CTrace evolved from a Unix-based tool I wrote several years ago. That version in turn evolved from various logging mechanisms found in systems-level daemons. What I've tried to do is extract the best of these older versions and add on a Mac flavor.

The *CTrace* class consists of three major elements — a control mechanism through which you submit messages for conditional logging, a circular buffer containing your logged messages, and a display window that gives you a view of the buffer.

The logging control mechanism, a subclass of *CDocument*, uses a bit mask that defines up to 32 event categories. I've included some basic categories — errors, warnings, general information, and function entry/exit — and you can add your own to fit your application. Your application submits requests to trace with a call like that in Example 1.

When *Trace( )* receives this message, it compares the mask passed as a parameter (in this example, *T_INFO*) with its internally stored mask. If you have enabled that particular category, *Trace( )* writes your data (along with a time stamp) into the log buffer. Otherwise, it happily ignores the request and returns.

The log buffer, based on the Think C class *CList*, stores some finite number of text strings. When it reaches the predefined limit, it begins overwriting the oldest entries with the newest. A scan of the list will always show the contents from the earliest entry to the most recent entry.

The display window shows the con-

Dr. Dobb's JOURNAL
SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

PUBLISHER *Peter Hutchinson*

EDITORIAL
EDITOR-IN-CHIEF *Jonathan Erickson*
MANAGING EDITOR *Monica E. Berg*
TECHNICAL EDITORS *Michael Floyd, Ray Valdes, David Betz*
ASSOCIATE MANAGING EDITOR *Janna Custer*
PRODUCTION EDITOR *Tami Zemel*
CONTRIBUTING EDITORS *Al Stevens, Jeff Duntemann, Tom Genereaux, Andrew Schulman, Ray Duncan*
COPY EDITORS *Pamela Dillehay, Nan Fornal*
EDITOR-AT-LARGE *Michael Swaine*

ART/PRODUCTION
ART/PRODUCTION DIRECTOR *Larry L. Clay*
ART DIRECTOR *Michael Hollister*
PRODUCTION SUPERVISOR *Amy Shulman Lesovoy*
TYPOGRAPHERS *Charlene Carpentier, Margaret Anderson*
COVER PHOTOGRAPHER *Michael Carr*

CIRCULATION
DIRECTOR OF CIRCULATION *Maureen Kaminski*
CIRCULATION MANAGER *Randy Robertson*
CIRCULATION PLANNING MANAGER *Manny Sawit*
DIRECT MARKETING MANAGER *Francesca Davies*
SINGLE COPY SALES DIRECTOR *Sarah Forsman*
FULFILLMENT MANAGER *Anne Jean*
DIRECT MARKETING COORDINATOR *Susan Bauman*
PROMOTION COORDINATOR *Philip Tsang*

ADMINISTRATION
VICE PRESIDENT OF FINANCE *Kate Deschamps*
CONTROLLER *Mary Collopy*
CREDIT MANAGER *Betty Arsene*
ACCOUNTING SUPERVISOR *Renate Kernke*
ACCOUNTS RECEIVABLE *Wendy Ho*
ACCOUNTS PAYABLE *LuAnn Rocklewitz*

MARKETING/ADVERTISING
ASSOCIATE PUBLISHER *Karla Spormann*
ADVERTISING COORDINATOR *Laura Stack Pullen*
MARKETING ASSISTANT *Sara Noah Ruddy*
ACCOUNT MANAGERS *see page 176*

M&T PUBLISHING INC.
CHAIRMAN OF THE BOARD *Otmar Weber*
DIRECTOR *C. F. von Quadt*
PRESIDENT *Laird Foshay*
VICE PRESIDENT OF PUBLISHING *William P. Howard*

DR. DOBB'S JOURNAL (USPS 307690) is published monthly by M&T Publishing, Inc., 501 Galveston Dr., Redwood City, CA 94063; 415-366-3600. Second-class postage paid at Redwood City and at additional entry points.

ARTICLE SUBMISSIONS: Send manuscripts and disk (with article, listings, and letter to the editor) to the associate managing editor 415-366-3600.

***DDJ* ON COMPUSERVE:** Type GO DDJ.

SUBSCRIPTION: $29.97 for 1 year; $56.97 for 2 years. Foreign orders must be prepaid in U.S. funds drawn on a U.S. bank. Canada and Mexico: $45.00 per year. All other foreign: $70.00 per year.

POSTMASTER: Send address changes to *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. **ISSN 1044-789X**

CUSTOMER SERVICE: For subscription questions, call toll-free 800-456-1215 (U.S. and Canada). For subscription orders or change of address call 303-447-9330 (all other countries) or write *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. For book/software orders call 800-533-4372 (in California 800-356-2002).

FOREIGN NEWSSTAND DISTRIBUTOR: Worldwide Media Service Inc., 115 E. 23rd St., New York, New York 10010; 212-420-0588. FAX 415-364-8630.

Entire contents copyright ©1990 by M&T Publishing, Inc., unless otherwise noted on specific articles. All rights reserved.

The Audit Bureau

(continued from page 44)

tents of the circular buffer. Because the buffer can potentially contain thousands of messages, the display window uses a subclass of the *CPanorama* class enclosed within a *CWindow*. Hence, you can scroll to any part of the buffer, you can grow and shrink the window, and you can hide the window when you don't need it. The buffer and window operate independently of each other, so even when you have hidden the window, logging to the buffer continues. The window looks like the one shown in Figure 1. You enable and disable categories via a modal dialog conjured up from a command on the Trace menu. It presents you with a dialog box like the one shown in Figure 2. You can set the trace mask at any time. For example, you may want to leave all categories disabled until you reach a point where you want to examine a particular string of events. Just before starting the transaction, summon up the dialog, make the appropriate changes, and continue.

The Gory Details

I won't dwell too much on the implementation of the code — it contains a plethora of comments and is built out of standard Think C core classes — but I do want to describe a bit of its family heritage so that you can understand how to include CTrace in your programs. The tool consists of three classes, *CTrace* (see CTrace.h in Listing One, page 116 and CTrace.c in Listing Two, page 116), *CLogPanorama* (see CLogPanorama.h in Listing Three, page 118, and CLogPanorama.c in Listing Four, page 118), and *CLogList* (see Listing Five, page 120, for CLogList.h and Listing Six, page 120, for CLogList.c). Of these, *CTrace* is specific to this tool, while *CLogPanorama* and *CLogList* are fairly generic.

CTrace is a subclass of the *CDocument* class that contains two local instance variables:

- *currMask*, the bit mask describing which categories are enabled and which are disabled.
- *itsLogPanorama*, a *CLogPanorama* that displays the log messages.

The class defines five local methods:

- *ITrace( )* initializes the object to contain some maximum number of log

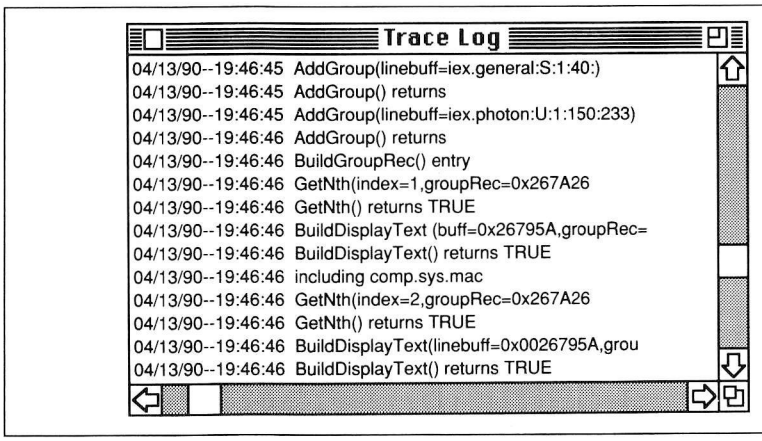


Figure 1: *CTrace message logging window*

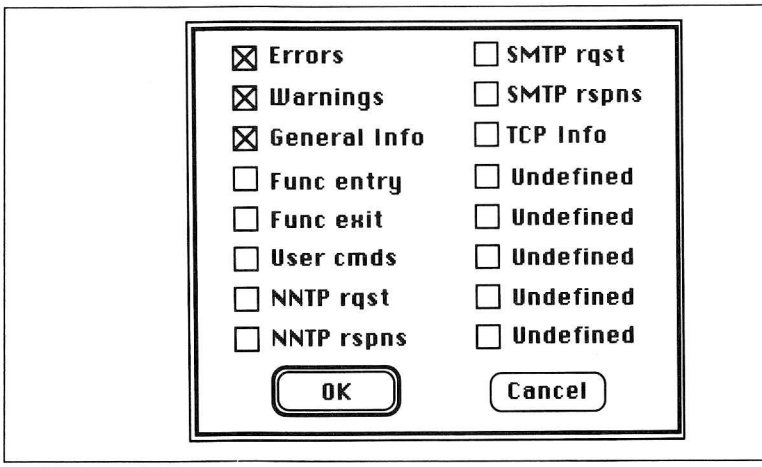


Figure 2: *CTrace dialog box*

(continued from page 46)

records.

- *ToggleTraceWindow( )* shows and hides *itsWindow* (*itsWindow* is a *CWindow* object inherited from *CDocument*).
- *SetTraceMask( )* presents a dialog for enabling and disabling individual trace mask bits.
- *Trace( )* receives a user message and category from the application and conditionally adds it to its buffer and window. The class defines an external, *gTrace*, which points to the class, so that code can reference this method directly as *gTrace->Trace( )* instead of using several levels of indirection via the application.
- *IsItVisible( )* returns the state of *itsWindow* so that the application can update its menus appropriately.

It overrides three inherited methods:

- *UpdateMenus( )* disables some of the non-applicable *CDocument* methods.
- *DoSaveAs( )* copies the buffer contents to a file.
- *Close( )* hides the window but does not actually close the *CTrace* document.

CLogPanorama is a subclass of the *CPanorama* class that contains a single, local instance variable:

- *itsLogList*, a *CLogList* circular buffer that contains the logged messages.

It defines two local methods:

- *ILogPane* initializes the panorama and installs it in the owner's window (in this case, CTrace's *itsWindow*).
- *AddString* inserts an ASCII string into *itsLogList*. When the window is visible, it also draws the string into the panorama and does whatever is required to make sure that the newly displayed line is visible. This involves a bit of special case code; refer to the inline comments for more elaborate details.

The class overrides one of its inherited methods using *Draw( )* to draw the visible portion of the buffer text into the panorama.

CLogList, based on *CList*, implements a circular buffer. It contains *maxRec*, a single, local instance variable that defines the maximum number of records allowed in the list and five local methods:

- *ILogList( )* initializes the list object.
- *AddString( )* appends the string to the end of the list. If the list has grown to its maximum, the method also deletes the first record in the list. Internally, the method allocates buffer space dynamically using handles on an as-needed basis.
- *GetString( )* returns a copy of a particular entry in the list.
- *GetMaxRecordCount( )* returns *maxRec*.

Finally, *CLogList* overrides the inherited method using *Dispose( )* to delete all of the records in the list.

Using Trace

Like most classes, CTrace requires a bit of preparation to use in your program.

First, add the resources in Table 1 to your application (you may change the resource IDs as required, but be sure to update the references in the header files appropriately). Within your code, you'll need to add an instance like that in Example 2 of the *CTrace* class to your application class.

In your *IApplication( )* method initialize *itsTraceLog* with the required number of records. For example, the code in Example 3 prepares CTrace to accept 1000 entries before wrapping. In your application's *UpdateMenus( )* method, add the statements in Example 4. Finally, add the commands in Example 5 to your application's *DoCommand( )* method.

To use CTrace effectively, you'll need to define masks for your logging categories. As you can see from the CTrace.h listing, I've defined five trace masks:

- *T_ERROR* for logging serious errors that may prove fatal.
- *T_WARNING* for logging problems that are probably not fatal.
- *T_INFO* for logging valuable runtime information.
- *T_FUNC_IN* for logging function or method entry (I usually include parameters passed into the function as arguments to *Trace( )*).
- *T_FUNC_OUT* for logging exit from functions or methods (I usually include the return value, if applicable, as an argument to *Trace( )*).

(continued on page 52)

Resource ID	Type	Description
2000	MENU	dtitled Trace, with the items *Show* (command 2000) and *Mask* (command 2001).
2000	DITL	buttons *OK* and *Cancel* (items 1 and 2), checkboxes *Errors, Warnings, Info, Func* entry, and *Func* exit (items 3 through 7), plus 27 additional checkboxes (items 8 through 34) titled *Undefined* and marked as disabled.
2000	DLOG	coordinates (14,34),(508,286), *procID 1*, marked as visible, and linked to DITL 2000.
2000	WIND	titled *Trace Log* at coordinates (16,52)(330,220), *procID 8*, marked not visible, with the *goAwayFlag* enabled.
1	MBAR	add menu res ID 2000

Table 1: *Added/modified resources*

```
gTrace->Trace (T_INFO, "Added '%s' to the display list",
subject);
```

Example 1: *Request to trace*

```
struct CMyApp : CApplication
        {
        CTrace  *itsTraceLog;
        ...
        };
```

Example 2: *Adding* CTrace *to* CApplication

```
itsTraceLog = new (CTrace);
itsTraceLog->ITrace (1000);
```

Example 3: *Initializing CTrace*

```
gBartender->EnableMenu (TRACE_MENU_ID);
gBartender->EnableCmd (TRACE_MENU_SHOW);
gBartender->EnableCmd (TRACE_MENU_MASK);
if (itsTraceLog->IsItVisible())
          gBartender->CheckMarkCmd (TRACE_MENU_SHOW, TRUE);
else
          gBartender->CheckMarkCmd (TRACE_MENU_SHOW, FALSE);
```

Example 4: *Additions to* UpdateMenus( ) *method*

(continued from page 48)

You can add your mask definitions directly to the CTrace.h file, or you may want to define them in a separate header. In any case, define each as a unique long word with one bit set as shown in Example 6. You'll also want to edit the tracemask DITL resource to put meaningful names on the set mask dialog box. A couple of comments may be appropriate here. First, the mask can contain up to 32 categories, but your application may not require this many. You may want to resize and rearrange the dialog box to show only those categories that you have defined. Note that DITL item #3 corresponds to mask 0x000000001, DITL #4 to mask 0x00000002, and so on. Also note that the dialog logic disables access to any items titled "Undefined." This prevents you from enabling categories for which there is no corresponding mask.

At this point, you can begin embedding log requests in your code. I have no great advice to give you with regard to when and where to place *Trace( )* statements in your code. I usually put them at the entry and exit points in functions, and, depending on the complexity of the code, before or after vital actions. And you should, of course, include them in your error checking logic.

One note with regard to the content of your log messages — the method uses *varargs*. Hence, in theory you can have as many arguments as you wish. Internally, however, the method has some self-preservation safeguards — if you try to form a message longer than 200 characters, it will truncate your message back to this limit.

```
case TRACE_MENU_SHOW :
          itsTraceLog->ToggleTraceWindow();
          break;
case TRACE_MENU_MASK :
          itsTraceLog->SetTraceMask();
          break;
```

Example 5: *Additions to* DoCommand( ) *method*

```
#define T_PICT_REDRAW   (0x00002000L) /* screen refresh logging */
```

Example 6: *Adding mask definitions*

Hide It or Remove It?

Once you've completed all of the coding and debugging for your program, you'll no doubt want to hide CTrace from the end-user. You could run through all of your code and purge all references to CTrace, but it's really much simpler to edit the MBAR resource so that the Trace menu entry doesn't show up on the menu bar. And, because the default trace mask has no categories enabled, you won't be using up much additional memory (remember that *CLogList* allocates buffer space dynamically). You could make an argument that all of those *Trace( )* calls will add unnecessary overhead to your program; I won't disagree that it adds some overhead, but because *Trace( )* returns immediately if it doesn't see a match between the passed mask and its internal mask, you're really looking at only a handful of machine instructions per call.

Keeping CTrace in the wings also enhances your support capabilities if you have friendly users — if some anom-

(continued on page 55)

(continued from page 52)
aly arises in the field, CTrace is only a *ResEdit* away.

Customizing CTrace

CTrace is ready for use out of the box. This is not to say that it doesn't have room for improvement. A short list of possible extensions includes the following:

- When the *CTrace* methods allocate internal memory, they make the rash assumption that memory will be available. My own version of CTrace includes a call to *CheckAllocation( )* around every *NewHandle( )*, but I left this out of the version described here because you probably have your own strategies for dealing with low memory.
- A clever macro could perform the mask compare inline and save the expense of unnecessary function calls. I'm more pragmatic than clever, so I haven't spent the time developing such a macro.
- The log pane uses simple QuickDraw *DrawString( )* commands to place text into the panorama. This limits the number of records you can store in the buffer to about 2700 records. If this proves insufficient, you may choose to modify the methods used by *CLogPanorama* to circumvent this limit.
- One of the predecessors of this utility embedded the file and the line number of the calling program as part of the logged message. Think C supports the necessary macros (*_FILE_* and *_LINE_*, respectively), but I chose not to include them in this version. You can certainly add them to your version.
- While CTrace supports the *DoSaveAs( ) CDocument* method, it does so only on request. A nice extension to CTrace may include writing the log message to a file as well as the *CLogPanorama*. This may come in handy if your programs have the nasty habit of leaving your Mac in some altered state!
- Finally, while I've done nothing to inhibit use of the inherited *DoPrint( )* method, I've also done nothing to enhance it. If you have a need to produce pretty, formatted printouts of the trace log, you can override the inherited *PrintPageOfDoc( )* method.

DDJ

(Listings begin on page 116.)

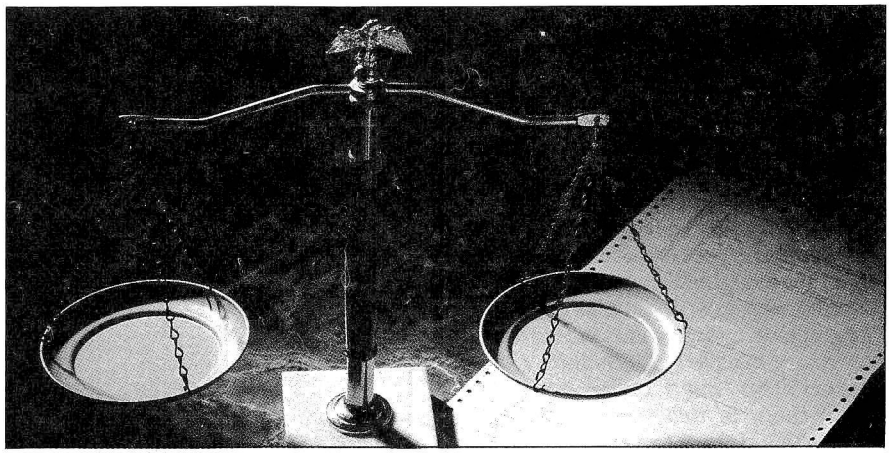
Vote for your favorite feature/article.
Circle Reader Service **No. 4.**

Software Patents

Is this the future of programming?

by The League for Programming Freedom

Software patents threaten to devastate America's computer industry. Newly granted software patents are being used to attack companies such as Lotus and Microsoft for selling programs that they have independently developed. Soon new companies will be barred from the software arena — most major programs will require licenses for dozens of patents, and this will make them infeasible. This problem has only one solution: Software patents must be eliminated.



The Patent System and Computer Programs

The Framers of the United States Constitution established the patent system so that inventors would have an incentive to share their inventions with the general public. In exchange for divulging an invention, the patent grants the inventor a 17-year monopoly on the use of the invention. The patent holder can license others to use the invention, but may also refuse to do so. Independent reinvention of the same technique by others does not give them the right to use it.

Patents do not cover specific computer programs; instead, they cover particular techniques that can be used to build programs, or particular features that programs can offer. Once a technique or feature is patented, it may not be used in a program without the permission of the patent holder — even if it is implemented in a different way. Since a program typically uses many techniques and provides many features, it can infringe many patents at once.

Until recently, patents were not used in the software field. Software developers copyrighted individual programs or made them trade secrets. Copyright was traditionally understood to cover the implementation details of a particular program; it did not cover the features of the program, or the general methods used. And trade secrecy, by definition, could not prohibit any development work by someone who did not know the secret.

On this basis, software development was extremely profitable and received considerable investment, without any prohibition on independent software development. But this scheme of things is no more. A few U.S. software patents were granted in the early 1980s, stimulating a flood of applications. Now many patents have been approved and the rate is accelerating. Many programmers are unaware of the change and do not appreciate the magnitude of its effects. Today the lawsuits are just beginning.

Absurd Patents

The Patent Office and the courts have had a difficult time with computer software. The Patent Office refused until recently to hire computer science graduates as examiners, and in any case does not offer competitive salaries for the field. Patent examiners are often ill-prepared to evaluate software patent applications to determine if they represent techniques that are widely known or obvious — both of which are grounds

This article is a position paper of the League for Programming Freedom, an organization opposed to software patents and interface copyrights and whose members include, among others, Marvin Minsky, John McCarthy, and Robert S. Boyer. Richard Stallman and Simson Garfinkel helped prepare this article for publication. You can contact the League through Internet mail (league@prep.ai.mit.edu) *or at 1 Kendal Square #143, P.O. Box 9171, Cambridge, MA 02139.*

(continued from page 56)
for rejection. Their task is made more difficult because many commonly used software techniques do not appear in the scientific literature of computer science: Some seemed too obvious to publish while others seemed insufficiently general.

Computer scientists know many techniques that can be generalized to widely varying circumstances. But the Patent Office seems to believe that each separate use of a technique is a candidate for a new patent. For example, Apple has been sued because the HyperCard program allegedly violates patent number 4,736,308, a patent that covers displaying portions of two or more strings together on the screen — effectively scrolling with multiple subwindows. Scrolling and subwindows are well-known techniques, but combining them is apparently illegal.

The granting of a patent by the Patent Office carries a presumption in law that the patent is valid. Patents for well-known techniques that were in use many years before the patent application have been upheld by federal courts.

For example, the technique of using *exclusive-or* to write a cursor onto a screen is both well-known and obvious. (Its advantage is that another identical *exclusive-or* operation can be used to erase the cursor without damaging the other data on the screen.) This technique can be implemented in a few lines of a program, and a clever high-school student might well reinvent it. But it is covered by patent number 4,197,590, which has been upheld twice in court even though the technique was used at least five years before the patent application. Cadtrak, the company that owns this patent, collects millions of dollars from large computer manufacturers.

English patents covering customary graphics techniques, including airbrushing, stenciling, and combination of two images under control of a third, were recently upheld in court, despite the testimony of the pioneers of the field that they had developed these techniques years before. (The corresponding United States patents, including 4,633,416 and 4,602,286, have not yet been tested in court, but they probably will be soon.)

All the major developers of spreadsheet programs have been threatened on the basis of patent 4,398,249, covering "natural order recalc," the recalculation of all the spreadsheet entries that are affected by the changes the user makes, rather than recalculation in a fixed order. Currently Lotus alone is being sued, but a victory for the plaintiff in the case would leave the other developers little hope. (The League for Programming Freedom has found *prior art* that may defeat this patent, but this is not assured.)

Nothing protects programmers from accidentally using a technique that is

(continued on page 62)

What You Can Do

The League for Programming Freedom is a grass-roots organization of programmers and users opposing software patents and interface copyrights. (The League is not opposed to copyright on individual programs.) Annual dues for individual members are $42 for employed professionals, $10.50 for students, and $21 for others. We appreciate activists, but members who cannot contribute their time are also welcome.

To contact the League, phone 617-243-4091, send Internet mail to:
league@prep.ai.mit.edu
or write to:

The League for Programming Freedom
1 Kendall Square #143
P.O. Box 9171
Cambridge, MA 02139

In the United States, another way to help is to write to Congress. You can write to your own representatives, but it may be even more effective to write to the subcommittees that consider such issues:

House Subcommittee on Intellectual Property
2137 Rayburn Bldg.
Washington, DC 20515

Senate Subcommittee on Patents, Trademarks and Copyrights
United States Senate
Washington, DC 20510

To write your own representatives, use the following addresses:

Senator *name*
United States Senate
Washington, DC 20510

Representative *name*
House of Representatives
Washington, DC 20515

(continued from page 58)
patented — and then being sued for it. Taking an existing program and making it run faster may also make it violate half a dozen patents that have been granted, or are about to be granted.

Even if the Patent Office learns to understand software better, the mistakes it is making now will follow us into the next century, unless Congress or the Supreme Court intervenes to declare these patents void.

However, this is not the whole of the problem. Computer programming is fundamentally different from the other fields that the patent system previously covered. Even if the patent system were to operate "as intended" for software, it would still obstruct the industry it is supposed to promote.

What is "Obvious"?

The patent system will not grant or uphold patents that are judged to be obvious. However, the system interprets the word "obvious" in a way that might surprise computer programmers. The standard of obviousness developed in other fields is inappropriate for software.

Patent examiners and judges are accustomed to considering even small, incremental changes as deserving new patents. For example, the famous Polaroid vs. Kodak case hinged on differences in the number and order of layers of chemicals in a film — differences between the technique Kodak was using and those described by previous, expired patents. The court ruled that these differences were unobvious.

Computer scientists solve problems quickly because the medium of programming is tractable. They are trained to generalize solution principles from one problem to another. One such generalization is that a procedure can be repeated or subdivided. Programmers consider this obvious — but the Patent Office did not think that it was obvious when it granted the patent on scrolling multiple strings as described above.

Cases such as this cannot be considered errors. The patent system is functioning as it was designed to do — but with software, it produces outrageous results.

Patenting What is too Obvious to Publish

Sometimes it is possible to patent a technique that is not new precisely because it is obvious — so obvious that no one would have published a paper about it.

For example, computer companies distributing the free X Window System (developed by MIT) are now being threatened with lawsuits by AT&T over

(continued on page 65)

(continued from page 62)
patent number 4,555,775, covering the use of "backing store." This technique is used when there are overlapping windows; the contents of a window that is partly hidden are saved in off-screen memory, so they can be put back quickly on the screen if the obscuring window disappears (as often happens).

The technique of backing store was used in an earlier MIT project, the Lisp Machine System, before AT&T applied for the patent. The Lisp Machine developers published nothing about this detail at the time, considering it too obvious. It was mentioned years later when the programmers' reference manual explained how to turn it on and off.

The Lisp Machine was the first computer to use this technique only because it had a larger memory than earlier machines that had window systems. Prior window system developers must have dismissed the idea because their machines had insufficient memory space to spare any for this purpose. Improvements in memory chips made development of backing store inevitable.

Without a publication, the use of backing store in the Lisp Machine System may not count as *prior art* to defeat the patent. So the AT&T patent may stand, and MIT may be forbidden to continue using a method that MIT used before AT&T.

The result is that the dozens of companies and hundreds of thousands of users who accepted the software from MIT with the understanding that it was free are now faced with possible lawsuits. (They are also being threatened with Cadtrak's *exclusive-or* patent.) The X Window project was intended to develop a window system that all developers could use freely. This public service goal seems to have been thwarted by patents.

Computer scientists are trained to generalize solution principles from one problem to another

Why Software is Different

Software systems are much easier to design than hardware systems of the same number of components. For example, a program of 100,000 components might be 50,000 lines long and could be written by two good programmers in a year. The equipment needed for this costs less than $10,000; the only other cost would be the programmers' living expenses while doing the job. The total investment would be less than $100,000. If done commercially in a large company, it might cost twice that. By contrast, an automobile typically contains under 100,000 components; it requires a large team and costs tens of millions of dollars to design.

And software is also much cheaper to manufacture: Copies can be made easily on an ordinary workstation costing under $10,000. To produce a hardware system often requires a factory costing tens of millions of dollars.

Why is this? A hardware system has to be designed using real components. They have varying costs; they have limits of operation; they may be sensitive to temperature, vibration, or humidity; they may generate noise; they drain power; they may fail either momentarily or permanently. They must be physically assembled in their proper places, and they must be accessible for replacement in case they fail.

Moreover, each of the components in a hardware design is likely to affect the behavior of many others. This greatly

complicates the task of determining what a hardware design will do: Mathematical modeling may prove wrong when the design is built.

By contrast, a computer program is built out of ideal mathematical objects whose behavior is defined, not modeled approximately, by abstract rules. When an *if* statement follows a *while* statement, there is no need to study whether the *if* statement will draw power from the *while* statement and thereby distort its output, nor whether it could overstress the *while* statement and make it fail.

Despite being built from simple parts, computer programs are incredibly complex. The program with 100,000 parts is as complex as an automobile, though far easier to design.

While programs cost substantially less to write, market, and sell than automobiles, the cost of dealing with the patent system will not be less. The same number of components will, on the average, involve the same number techniques that might be patented.

The Danger of a Lawsuit

Under the current patent system, a software developer who wishes to follow the law must determine which patents a program violates and negotiate with each patent holder a license to use that patent. Licensing may be prohibitively expensive, as in the case when the patent is held by a competitor. Even "reasonable" license fees for several patents can add up to make a project infeasible. Alternatively, the developer may wish to avoid using the patent altogether; but there may be no way around it.

The worst danger of the patent system is that a developer might find, after releasing a product, that it infringes one or many patents. The resulting lawsuit and legal fees could force even a medium-size company out of business.

Worst of all, there is no practical way for a software developer to avoid this danger — there is no effective way to find out what patents a system will infringe. There is a way to try to find out — a patent search — but searches are unreliable and in any case too expensive to use for software projects.

Patent Searches are Prohibitively Expensive

A system with 100,000 components can use hundreds of techniques that might already be patented. Since each patent search costs thousands of dollars, searching for all the possible points of danger could easily cost over a million. This is far more than the cost of writing the program.

The costs don't stop there. Patent

applications are written by lawyers for lawyers. A programmer reading a patent may not believe that his program violates the patent, but a federal court may rule otherwise. It is thus now necessary to involve patent attorneys at every phase of program development.

Yet this only reduces the risk of being sued later — it does not eliminate the risk. So it is necessary to have a reserve of cash for the eventuality of a lawsuit.

When a company spends millions to design a hardware system, and plans to invest tens of millions to manufacture it, an extra million or two to pay for dealing with the patent system might be bearable. However, for the inexpensive programming project, the same extra cost is prohibitive. Individuals and small companies especially cannot afford these costs. Software patents will put an end to software entrepreneurs.

Patent Searches are Unreliable

Even if developers could afford patent searches, these are not a reliable method of avoiding the use of patented techniques. This is because patent searches do not reveal pending patent applications (which are kept confidential by the Patent Office). Since it takes several years on the average for a software patent to be granted, this is a serious problem: A developer could begin designing a large program after a patent has been applied for, and release the program before the patent is approved. Only later will the developer learn that distribution of the program is prohibited.

For example, the implementors of the widely used public domain data compression program *compress* followed an algorithm obtained from *IEEE Computer* magazine. They and the user community were surprised to learn later that patent number 4,558,302 had been issued to one of the authors of the article. Now Unisys is demanding royalties for using this algorithm. Although the program is still in the public domain — using it means risking a lawsuit.

The Patent Office does not have a workable scheme for classifying software patents. Patents are most frequently classified by end results, such as "converting iron to steel," but many patents cover algorithms whose use in a program is entirely independent of the purpose of the program. For example, a program to analyze human speech might infringe the patent on a speedup in the Fast Fourier Transform; so might a program to perform symbolic algebra (in multiplying large numbers); but the category to search for such a patent would be hard to predict.

(continued on page 70)

(continued from page 67)

You might think it would be easy to keep a list of the patented software techniques, or even simply remember them. However, managing such a list is nearly impossible. A list compiled in 1989 by lawyers specializing in the field omitted some of the patents mentioned in this article.

Obscure Patents

When you imagine an invention, you probably think of something that could be described in a few words, such as "a flying machine with fixed, curved wings" or "an electrical communicator with a microphone and a speaker." But most patents cover complex detailed processes that have no simple descriptions — often they are speed-ups or variants of well-known processes that are themselves complex.

Most of these patents are neither obvious nor brilliant; they are obscure. A capable software designer will "invent" several such improvements in the course of a project. However, there are many avenues for improving a technique, so no single project is likely to find any given one.

For example, IBM has several patents (including patent 4,656,583) on workmanlike, albeit complex, speed-ups for well-known computations performed by optimizing compilers, such as register coloring and computing the available expressions.

Patents are also granted on combinations of techniques that are already widely used. One example is IBM patent 4,742,450, which covers "shared copy-on-write segments." This technique allows several programs to share the same piece of memory that represents information in a file; if any program writes a page in the file, that page is replaced by a copy in all of the programs, which continue to share that page with each other but no longer share with the file.

Shared segments and copy-on-write have been used since the 1960s. This particular combination may be new as a specific feature, but is hardly an invention. Nevertheless, the Patent Office thought that it merited a patent, which must now be taken into account by the developer of any new operating system.

Obscure patents are like land mines: Other developers are more likely to reinvent these techniques than to find out about the patents, and then they will be sued. The chance of running into any one of these patents is small, but they are so numerous that you cannot go far without hitting one. Every basic technique has many variations, and a small set of basic techniques can be combined in many ways. The patent office has now granted more than 2000 software patents — 700 in 1989 alone. We can expect the pace to accelerate. In ten years, programmers will have no choice but to march on blindly and hope they are lucky.

Patent Licensing has Problems, too

Most large software companies are trying to solve the problem of patents by getting patents of their own. Then they hope to cross-license with the other large companies that own most of the patents, so they will be free to go on as before.

While this approach will allow companies such as Microsoft, Apple, and IBM to continue in business, it will shut new companies out of the field. A future start-up, with no patents of its own, will be forced to pay whatever price the giants choose to impose. That price might be high: Established companies have an interest in excluding future competitors. The recent Lotus lawsuits against Borland International and the Santa Cruz Operation (although involving an extended idea of copyright rather than patents) show how this can work.

Even the giants cannot protect themselves with cross-licensing from companies whose only business is to buy patents and then threaten to sue. For example, the New York-based Refac Technology Development Corporation, a company that represents Forward Reference Systems (owners of the patent for *natural order recalc*), recently sued Lotus Corporation. *Natural order recalc* is Refac's first foray into the software patent arena; for the past 40 years, the company has negotiated licenses in the fastener and electronic component industries. The company employs no programmers or engineers.

Refac is demanding — in the neighborhood of — five percent of sales of all major spreadsheet programs. If a future program infringes on 20 such patents — and this is not unlikely, given the complexity of computer programs and the broad applicability of many patents — the combined royalties could exceed 100 percent of the sales price.

The Fundamental Question

According to the Constitution of the United States, the purpose of patents is to "promote the progress of science and the useful arts." Thus, the basic question at issue is whether software patents, supposedly a method of encouraging software progress, will truly do so, or will retard progress instead.

So far we have explained the ways

in which patents will make ordinary software development difficult. But what of the intended benefits of patents: More invention, and more public disclosure of inventions? To what extent will these actually occur in the field of software?

There will be little benefit to society from software patents because invention in software was already flourishing before software patents, and inventions were normally published in journals for everyone to use. Invention flourished so strongly, in fact, that the same inventions were often found again and again.

In Software, Independent Reinvention is Commonplace

A patent is an absolute monopoly; everyone is forbidden to use the patented process, even those who reinvent it independently. This policy implicitly assumes that inventions are rare and precious, because only in those circumstances is it beneficial.

The field of software is one of constant reinvention; as some people say, programmers throw away more "inventions" each week than other people develop in a year. And the comparative ease of designing large software systems makes it easy for many people to do work in the field. A programmer solves many problems in developing each program. These solutions are likely to be reinvented frequently as other programmers tackle similar problems.

The prevalence of independent reinvention negates the usual purpose of patents. Patents are intended to encourage inventions and, above all, the disclosure of inventions. If a technique will be reinvented frequently, there is no need to encourage more people to invent it; because some of the developers will choose to publish it (if publication is merited), there is no point in encouraging a particular inventor to publish it — not at the cost of inhibiting use of the technique.

Overemphasis of Inventions

Many analysts of the American and Japanese industry have attributed Japanese success at producing quality products to the fact that they emphasize incremental improvements, convenient features, and quality rather than noteworthy inventions.

It is especially true in software that success depends primarily on getting the details right. And that is most of the work in developing any useful software system. Inventions are a comparatively unimportant part of the job.

The idea of software patents is thus an example of the mistaken American preoccupation with inventions rather than products. And patents will further reinforce this mistake, rewarding not the developers who write the best software, but those who were first to file for a patent.

Impeding Innovation

By reducing the number of people engage in software development, software patents will actually impede innovation. Much software innovation comes from programmer's solving problems while developing software, not from projects whose specific purpose is to make invention and obtain patents. In other words, these innovations are byproducts of software development.

When patents make development more difficult, and cut down on development projects, they will also cut down on the byproducts of development — new techniques.

Could Patents Ever be Beneficial?

Although software patents are in general harmful to society as a whole, we do not claim that every single software patent is necessarily harmful. Careful study might show that under certain specific and narrow conditions (necessarily excluding the vast majority of cases) it is beneficial to grant

software patents.

Nonetheless, the right thing to do now is to eliminate all software patents as soon as possible, before more damage is done. The careful study can come afterward.

Clearly, software patents are not urgently needed by anyone, except patent lawyers. The prepatent software industry had no problem that was solved by patents; there was no shortage of invention and no shortage of investment. Complete elimination of software patents may not be the ideal solution, but it is close, and is a great improvement. Its very simplicity helps avoid a long delay while people argue about details. If it is ever shown that software patents are beneficial in certain exceptional cases, the law can be changed again at that time — if it is important enough. There is no reason to continue the present catastrophic situation until that day.

Software Patents are Legally Questionable

It may come as a surprise that the extension of patent law to software is still legally questionable. It rests on an extreme interpretation of a particular 1981 Supreme Court decision, Diamond vs. Deihr. (See "Legally Speaking" in *Communications of the ACM*, August 1990.)

Traditionally, the only kinds of processes that could be patented were those for transforming matter (such as for transforming iron into steel). Many other activities which we would consider processes were entirely excluded from patents, including business methods, data analysis, and "mental steps." This was called the "subject matter" doctrine.

Diamond vs. Deihr has been interpreted by the Patent Office as a reversal of this doctrine, but the court did not explicitly reject it. The case concerned a process for curing rubber — a transformation of matter. The issue at hand was whether the use of a computer program in the process was enough to render it unpatentable, and the court ruled that it was not. The Patent Office took this narrow decision as a green light for unlimited patenting of software techniques, and even for the use of software to perform specific well-known and customary activities.

Most patent lawyers have embraced the change, saying that the new boundaries of patents should be defined over decades by a series of expensive court cases. Such a course of action will certainly be good for patent lawyers, but it is unlikely to be good for software developers and users.

One Way to Eliminate Software Patents

We recommend the passage of a law to exclude software from the domain of patents. That is to say that, no matter what patents might exist, they would not cover implementations in software; only implementations in the form of hard-to-design hardware would be covered. An advantage of this method is that it would not be necessary to classify patent applications into hardware and software when examining them.

Many have asked how to define software for this purpose — where the line should be drawn. For the purpose of this legislation, software should be defined by the characteristics that make software patents especially harmful:

- Software is built from ideal infallible mathematical components, whose outputs are not affected by the components they feed into.
- Ideal mathematical components are defined by abstract rules, so that failure of a component is by definition impossible. The behavior of any system built of these components is likewise defined by the consequences of applying the rules step by step to the components.
- Software can be easily and cheaply copied.

Following this criterion, a program to compute prime numbers is a piece of software. A mechanical device de-

signed specifically to perform the same computation is not software, because mechanical components have friction, can interfere with each other's motion, can fail, and must be assembled physically to form a working machine.

Any piece of software needs a hardware platform in order to run. The software operates the features of the hardware in some combination, under a plan. Our proposal is that combining the features in this way can never create infringement. If the hardware alone does not infringe a patent, then using it in a particular fashion under control of a program should not infringe either. In effect, a program is an extension of the programmer's mind, acting as a proxy for the programmer to control the hardware.

Usually the hardware is a general-purpose computer, which implies no particular application. Such hardware cannot infringe any patents except those covering the construction of computers. Our proposal means that, when a user runs such a program into a general-purpose computer no patents other than those should apply.

The traditional distinction between hardware and software involves a complex of characteristics that used to go hand in hand. Some newer technologies, such as gate arrays and silicon compilers, blur the distinction because they combine characteristics associated with hardware with others associated with software. However, most of these technologies can be classified unambiguously for patent purposes, either as software or as hardware, using the criteria above. A few gray areas may remain, but these are comparatively small, and need not be an obstacle to solving the problems patents pose for ordinary software development. They will end up being treated as hardware, as software, or as something in between.

Fighting Patents One by One

Until we succeed in eliminating all patenting of software, we must try to overturn individual software patents. This is very expensive and can solve only a small part of the problem, but that is better than nothing.

Overturning patents in court requires *prior art*, which may not be easy to find. The League for Programming Freedom will try to serve as a clearing house for this information, to assist the defendants in software patent suits. This depends on your help. If you know about *prior art* for any software patent, please send the information to the League (see the accompanying text box.)

If you work on software, you can personally help prevent software patents by refusing to cooperate in applying for them. The details of this may depend on the situation.

Conclusion

Exempting software from the scope of patents will protect software developers from the insupportable cost of patent searches, the wasteful struggle to find a way clear of known patents, and the unavoidable danger of lawsuits.

If nothing is changed, what is now an efficient creative activity will become prohibitively expensive. The sparks of creativity and individualism that have driven the computer revolution will be snuffed out.

To picture the effects, imagine that each square of pavement on the sidewalk has an owner and that pedestrians must obtain individual licenses to step on particular squares. Think of the negotiations necessary to walk an entire block under this system. That is what writing a program will be like in the future if software patents continue.

DDJ

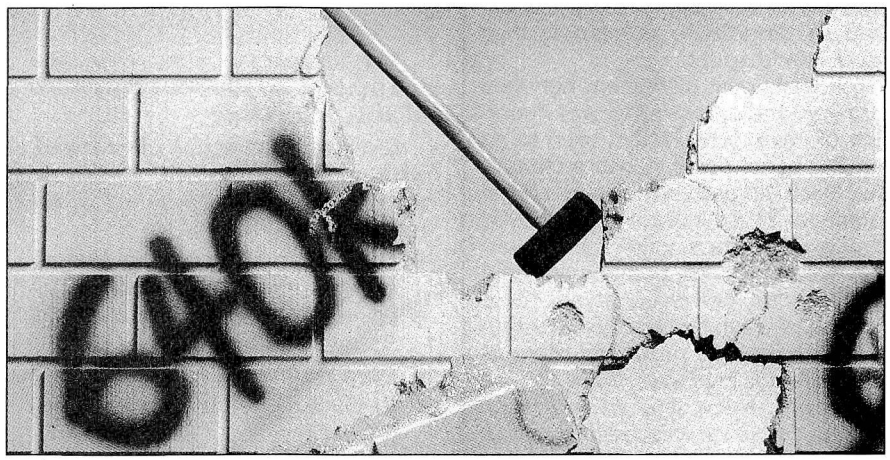
Vote for your favorite feature/article.
Circle Reader Service **No. 5.**

Roll your Own DOS Extender: Part II

Under the hood

Al Williams

Last month, I discussed 80*x*86 protected mode in general and presented the basics of PROT, the DOS extender described in this two-part article. In this installment, I'll examine both 80386 debugging and exceptions, then take you under the DOS extender's hood. Listings One through Three were covered in Part I. In this installment, I cover Listings Four through Seven: STACKS.INC (Listing Four, page 122) contains the stack segments; INT386.INC (Listing Five, page 122) is for 386 interrupt handling; TSS.INC (Listing Six, page 126) contains the task state segment definitions; and CODE16.INC (Listing Seven, page 126) is the 16-bit DOS entry/exit code.



Because most programs don't work right the first few hundred times you try them, PROT contains full debugging support. Of course, the 386 has many hardware debugging features built-in. Except for the single-step capability, these features are all available with PROT. In addition, EQUMAC.INC (see Listing Three in last month's installment) contains macros to set normal, conditional, and counter breakpoints. When a breakpoint or unexpected interrupt occurs, PROT will display a register and stack dump. You can also instruct PROT to dump a memory region along with the register dump.

Figure 3 shows the screen PROT displays when an unexpected interrupt occurs. This interrupt may be a breakpoint (INT 3) or an 80386 exception (refer to Table 4). Most likely, it will be INT 0DH, the dreaded general-protection interrupt.

PROT prints the display on the screen using *OUCH* and its related routines. This means that if you have changed the video mode or the page in your program, you may have to modify the *OUCH* routine to accommodate those changes. You could, for instance, modify *OUCH* to output to a printer. Be careful, however, not to use any DOS or BIOS routines in *OUCH*; you can't be sure what state the system will be in when *OUCH* is called.

The display is largely self-explanatory. PROT displays the registers and the interrupt number at the top of the screen. Below the registers, PROT displays a stack dump, starting with the location at ESP and continuing to the end of the stack segment. If the location *DUMP_SEL* is non-zero, PROT also prints *DUMP_CNT* bytes starting at *DUMP_SEL:DUMP_OFF*. Note that the *BREAKDUMP* and *NBREAKDUMP* macros (see Table 5 for complete list of macros to set breakpoints) automatically set these values for you. After printing this screen, PROT returns to DOS with an error code of 7FH.

Some 80386 exceptions push an error code on the stack. For these exceptions, that will be the first word on the stack. The next two words will be the value of CS:EIP at the time of the interrupt. In Figure 3, for example, the value of CS:EIP is 88H:2BH. The next word will be the flags (202H in Figure 3). The stack dump displays the protected-mode stack, even if a VM86 mode program was interrupted. If you need to look at your VM86 stack, you can use the memory dump feature. When an

Al is a systems engineer on the space station Freedom project for Jackson and Associates. Look for an expanded version of PROT in his book DOS: A Developer's Guide, *which will be available from M&T Books early in 1991. Al can be reached at 310 Ivy Glen Ct., League City, TX 77573, or via CompuServe at 72010,3574.*

(continued from page 74)
interrupt occurs during a VM86 program, the words following the flags on the stack are the SS, ES, DS, FS, and GS registers, in that order. Any segment register values on the stack will have their top 16 bits set to unpredictable values by the 386.

80386 Exceptions

The 80386 defines 15 different exceptions that it generates internally to indicate certain conditions. Table 4 summarizes all of the possible exceptions and why they occur. Exceptions fall into three classes: faults, traps, and aborts. A fault is restartable; the offending instruction hasn't executed yet, and CS:EIP points to the instruction that faulted. In a trap exception, the offending instruction has already executed. CS:EIP then points to the instruction that will execute *after* the offending instruction. Aborts are exceptions so severe that you will usually terminate the program generating them. Table 6 presents these exceptions in detail.

The only exception that doesn't fall into one of these categories is INT 1, the debug exception. Some debug exceptions are faults and some are traps. Remember, PROT ends the running program and does a debug dump when

(continued on page 80)

```
ES=0040     DS=0090     FS=0010     GS=0038
EDI=00000000 ESI=00000000 EBP=00000000 ESP=00000FF0 EBX=00000000
EDX=00000000 ECX=00000000 EAX=00000000 INT=03 TR=0070
Stack Dump:
0000002B 00000088 00000202
```

Figure 3: *PROT interrupt/breakpoint display*

Name	INT #	Type	Error Code	Possible causes
Divide error	0	FAULT	No	DIV or IDIV
Debug	1	N/A	No	Debug condition
Breakpoint	3	TRAP	No	INT 3
Overflow	4	TRAP	No	INTO
Bounds check	5	FAULT	No	BOUND
Bad opcode	6	FAULT	No	Illegal instruction
No 80x87	7	FAULT	No	ESC, WAIT with no coprocessor or when coprocessor was last used by another task
Double fault	8	ABORT	Yes (always 0)	Any instruction that can generate an exception
NPX overrun	9	ABORT	No	Any operand of an ESC that wraps around the end of a segment.
Invalid TSS	10	FAULT	Yes	JMP, CALL, IRET or interrupt
No segment	11	FAULT	Yes	Any reference to a not-present segment
Stack error	12	FAULT	Yes	Any reference with SS register
Gen'l protect	13	FAULT	Yes	Any memory reference including code fetches
Page fault	14	FAULT	Yes	Any memory reference including code fetches
NPX error	16	FAULT	No	ESC, WAIT

Table 4: *80386/80486 exceptions*

Macro	Description
BREAKPOINT	Execution will stop at this point, causing a debug dump.
BREAKON	Turns on conditional breakpoints. If followed by an integer number, that number of conditional breakpoints must execute before the breakpoint actually occurs.
BREAKOFF	Turns off conditional breakpoints.
NBREAKPOINT	Conditional breakpoint macro. Until a BREAKON command executes, this instruction will have no effect.
BREAKDUMP	Causes an unconditional breakpoint and dumps a region of memory following the stack dump.
NBREAKDUMP	Conditional breakpoint with the memory dump feature of BREAKDUMP added.

Table 5: *Macros used to set breakpoints*

(continued from page 76)
any unexpected exception occurs, unless you reprogram the interrupt routines to do differently.

Some exceptions push an error code on the stack. Exception 14 has a special error code; but all the other exceptions simply push the segment selector that caused the fault onto the stack. The error code 0 means that the fault was caused by either the null selector or a selector which the 80386 was unable to determine.

What Went Wrong?

Real-mode debuggers, such as CodeView, Turbo Debugger, or DEBUG, won't help you troubleshoot PROT pro-

Exception	Description
INT 0	Occurs if you ask the 80386 to divide by 0, or if a division gives a result that will not fit in the specified accumulator (AL, AX, or EAX).
INT 1	Vector for a multitude of debugging interrupts. Hardware data breakpoints, single-step breakpoints, and task-switch breakpoints are all considered traps. Hardware code breakpoints and general detect exceptions are both faults. Enabling any of these breakpoints requires manipulating the debug registers (DR0-DR7).
INT 3	Occurs when an INT 3 instruction executes. Convenient for placing breakpoints in your code.
INT 4	Check arithmetic operations for overflow. If an INTO instruction executes with the OF flag set, INT 4 occurs.
INT 5	The BOUND instruction generates this interrupt if it finds an array index outside of the array's limits. Real-mode programmers don't use the BOUND instruction because the ROM BIOS uses INT 5 for the print screen routine.
INT 6	Occurs when the 80386 reads an unknown opcode or when a legitimate instruction receives a bad operand.
INT 7	Occurs when: 1. A coprocessor instruction executed, but the system contains no coprocessor (as indicated by the EM flag in CR0); 2. A coprocessor instruction executed, but the last coprocessor operation occurred in another task. This exception is only generated when the MP flag in CR0 equals one. The second form of INT 7 is useful for maintaining the coprocessor's consistency when several different tasks are using the 80387.
INT 8	One exception occurred while trying to begin service for another exception. May require terminating the program.
INT 9	Occurs when a coprocessor instruction runs over the end of a segment.
INT 10	Occurs when a task switch occurs to an improperly formed TSS. Handling this exception properly requires interrupt pass through a task gate. PROT does not do this, however. If you plan to do extensive multitasking with PROT, you should supply your own INT 10 handler to simplify debugging.
INT 11	Occurs when the processor loads a not-present segment descriptor or uses a not-present gate descriptor.
INT 12	Indicates that the stack has over- or under-run, or that the stack segment register (SS) was loaded with a not-present segment.
INT 13	General protection fault is perhaps the most familiar interrupt for 80386 programmers. This covers any error not handled by the other exceptions.
INT 14	Occurs when a referenced memory page is absent, or a page's privilege is higher than the program that attempted to access it. This is the primary vehicle used to implement virtual memory on the 80386.
INT 16	Occurs when the 80387 detects an error.

Table 6: *80386 exceptions in detail*

```
.386P
SEGMENT   EXAMPLE PARA 'CODE32' USE32
          .
BACKWARD:
          .
          .
          .
          CMP EBX,EAX
          JA FORWARD          ; This jump is OK
          JB BACKWARD         ; This jump is improperly assembled
          .
          .
          .
FORWARD:
```

Example 4: *32-bit offset generation problems*

grams. When an exception display screen appears, you should note the CS:EIP (from the stack dump). By referring to the listing file generated by the assembler, you should be able to pinpoint the exact instruction that caused the exception.

Exceptions are commonly caused by referencing segment registers that contain zero (the null selector), addressing outside of a segment, or attempting to write into a code segment. Another common error stems from the linker's inability to generate 16-bit relative offsets. For instance, consider the code fragment in Example 4. This code makes two conditional jumps, one if the value in EBX is above EAX and the other if it is below.

Because the code contains the .386P (or .386) directive, all conditional jumps default to 32-bit relative jumps in a 32-bit segment. The first jump will be correct because it requires a positive offset less than 64K away. The second jump will probably cause a general-protection fault because the linker only generates 2 bytes of the 4-byte offset. For example, if the jump's offset is –10 (FFFFFFF6 hex), the linker will generate FFF6 hex, a 32-bit offset of 65,526. This is sure to cause an unexpected result. If you are lucky, the segment isn't that large and a general-protection fault occurs. Otherwise, the 386 will just jump to a new location. Sometimes, this location will not contain a valid instruction, which will cause an exception 6. Other times, your program will just start doing entirely unpredictable things. To prevent this, and for efficiency's sake, you should specify all jumps to be short, if possible. If not, you can use the *JCC32* macro provided in Listing Three (Part I) to generate proper 32-bit conditional jumps.

Exceptions fall into three classes: faults, traps, and aborts

Under the Hood

Most of the implementation of PROT is straight out of the Intel documentation. The program sets up the Global Descriptor Table (GDT), disables interrupts, and switches the machine into protected mode. In protected mode, PROT sets up the Interrupt Descriptor Table (IDT), reprograms the interrupt controllers, and reenables interrupts. PROT then calls the user's code. While running DOS or BIOS code, PROT emulates the *PUSHF, POPF, STI, CLI, INT,* and *IRET* instructions.

As mentioned earlier, software interrupts pose the single biggest problem for a DOS extender. The easy case occurs when a program calls an interrupt routine with *INT,* and the interrupt routine ends with *IRET.* Most of the DOS/BIOS calls, however, return information in the flag register. Because *IRET* will restore the flags, these calls have to use some method of overriding the old flags. From a DOS extender's point of view, the right thing to do is modify the flags on the stack. This causes *IRET* to restore the flags we want instead of the original ones.

Unfortunately, most of the system calls don't do this. One common method of sending flags back to the caller is to return using a *RETF 2* instruction instead of *IRET.* In real mode, this has the same effect as *IRET,* except for destroying the saved flags. Of course, a VM86 *RETF* instruction can't return to a protected-mode task, so a DOS extender has to find a way around this.

There is yet another way DOS handles flags: The absolute disk read and

(continued on page 82)

```
Case 1

        Normal INT/IRET
                :
                INT 10H         ; perform interrupt
                :
ISR:            :               ; Interrupt 10H service routine
                :
                IRET
Case 2
                INT/RETF 2
                :
                INT 10H         ; perform interrupt
                :
ISR:            :               ; Interrupt 10H service routine
                :
                RETF 2
Case 3
        INT/RETF (only used by INT 25H and 26H)
                :
                INT 10H         ; perform interrupt
                :
ISR:            :               ; Interrupt 10H service routine
                :
                RETF
Case 4
        PUSHF/FAR CALL
                :
                PUSHF           ; simulate interrupt
                CALL FAR ISR
                :
ISR:            :               ; Interrupt 10H service routine
                :
                IRET
Case 5
        PUSHF/PUSH ADDRESS/IRET
                :
                PUSHF           ; Jump to address TARGET
                PUSH SEG TARGET
                PUSH OFFSET TARGET
                IRET
                :
TARGET:         :               ; Destination of IRET
                :
-or-            :
                PUSHF           ; Simulate interrupt
                PUSH SEG RETAD
                PUSH OFFSET RETAD
                JMP FAR ISR
RETAD:          :
                :
ISR:            :               ; Interrupt routine
                :
                IRET
```

Figure 4: *Problem cases associated with software interrupts. Case 1, normal interrupt routine call and return; case 2, returning from an interrupt using FAR RETURN 2; case 3, returning from an interrupt using a FAR RETURN; case 4, simulating an interrupt with a FAR CALL; case 5, using IRET to transfer control to an arbitrary address.*

write interrupts (INT 25H and 26H) leave the original flags at the top of the stack. These interrupt routines accomplish this by executing *RETF*. This is similar to the previous case, and PROT handles it in the same way.

Another problem lies in the DOS and BIOS methods of calling interrupts. Most often, these routines use an *INT* instruction to start an interrupt routine. However, systems routines will sometimes push flags on the stack and then do a *FAR CALL*. In real mode, this is exactly equivalent to an *INT*. In protected mode, however, this will cause havoc in our attempts to emulate the *IRET* instruction. Luckily, the method we will use to handle the *RETF 2* return also suggests a solution to this problem.

Finally, the system routines sometimes pass control by pushing the flags and an address on the stack and then executing an *IRET* instruction. This resembles the *FAR CALL/IRET* case, and PROT handles it in the same manner. These problem cases are detailed with the code fragments in Figure 4.

```
qisr segment para 'CODE16' use16
        assume cs:qisr
    qiret:
        push 0
        push 0
        push 0
        iret
    qisr ends
```

Example 5: *QISR code for the VM86 mode segment*

The Seven Percent Solution

Our DOS extender must have a VM86 mode segment (I've called it QISR), that contains the code shown in Example 5. When the DOS extender detects an *INT* instruction being executed from real mode, it emulates it much as outlined in the Intel documentation. The 80386 places the actual flags and return address on the Privilege Level 0 (PL0) stack via the general-protection fault caused by the *INT*. The secret to managing DOS's odd interrupt handling lies in manipulating the VM86 stack. You can push a 16-bit copy of the flags on the VM86 stack, followed by the address of *qiret*. Then the DOS extender transfers control (in VM86 mode) to the real-mode interrupt handler.

When an *IRET* executes, another general-protection fault occurs. We can determine which case happened by examining the top of the VM86 stack. If the top three words of the stack are all zeros, we have found case 2 or 3 in Figure 4. If the address on the top of the stack is *qiret*'s, then the normal case (case 1) occurred. Any other address at the top of the stack indicates case 4 or 5.

Now that we know what case caused the IRET, what action does the DOS extender take? PROT uses the following logic:

In case 1, merge the 16-bit flags on the VM86 stack with the top 16 bits of the flags pushed on the PL0 stack during the *INT* handling. This may seem redundant, but the system routines may modify these flags, and the caller will expect the modified flags. Balance the VM86 stack and restore the return address from the PL0 stack; cases 2 and 3 are much the same as case 1, except that PROT restores the current flags (which are on the PL0 stack) instead of the flags on the VM86 stack. Cases 4 and 5 don't go through the *INT* logic just described. Therefore, the DOS extender must build an artificial PL0 stack frame that looks as though the *INT* logic executed earlier. The execution then continues as in case 1.

When a protected-mode program needs to call a DOS or BIOS service, it calls the *call86* routine. This routine uses PROT's *INT 30H* function to build a stack frame much like the one used in cases 4 and 5 and simulates the DOS (or BIOS) interrupt.

(continued from page 83)

Hardware Interrupts

INT 30H directly handles hardware interrupts. Because hardware interrupts can occur at any time, PROT examines the flags of the interrupted program. If the interrupted program is a VM86 program, PROT uses the VM86 stack to handle the interrupt. If the program is a protected-mode program, PROT switches to a special stack for hardware interrupt processing.

The PC's first interrupt controller is reprogrammed to generate different interrupts than normal. PROT translates these interrupts to the correct service routines. This prevents hardware interrupts from being confused with 80386 exceptions. Also, the interrupts from the second controller (on AT-style machines only) are reprogrammed to simplify constructing the interrupt table.

Future Directions

While PROT is a complete protected-mode environment, there are still some interesting enhancements you may want to try. For completeness, PROT should emulate *IRETD, PUSHFD,* and *POPFD* in VM86 mode. Adding a scheduler that would run off the PC's timer interrupt would make PROT more useful in multitasking applications. Additional support for memory paging would be helpful as well.

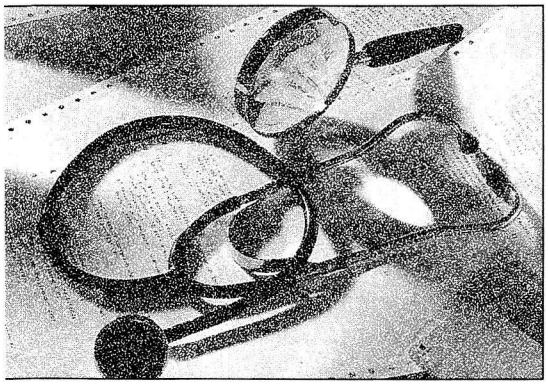
PROT's most noticeable shortcoming is it's inability to work with other 386-specific programs. Standards such as the Virtual Control Program Interface (VCPI) and the DOS Protected Mode Interface (DPMI) allow programs like PROT to share the 386 with other programs that also use the processor's special features. Adding VCPI or DPMI support to PROT should not be very difficult. The specification for VCPI is available, free of charge, from Phar Lap Software Inc., 60 Aberdeen Ave., Cambridge, MA 02138, 617-661-1510. The DPMI spec can be ordered free of charge from Intel Literature Sales, P.O. Box 58130, Santa Clara, CA 95052, 800-548-4725 (Intel part # 240763-001).

DOS extenders provide a way for today's developers to launch sophisticated applications now, and with a much larger potential market than any other option. By using PROT, you will be able to learn more about developing protected-mode software. The experience you gain will not only help you expand your DOS applications today, it will also give you a headstart on the operating systems of the future.

DDJ

(Listings begin on page 122.)

Vote for your favorite feature/article.
Circle Reader Service **No. 6.**



Programmer Tools For Actor 3.0

Data management and resource editing for Windows development

Marty Franz

A colleague of mine (who shall remain nameless) has a favorite aphorism: If a program is tough to write, it should be tough to use. Given this mindset, it should come as no surprise that he dislikes writing programs for Microsoft Windows, especially in C. I don't agree with this statement, but I do agree with its contrapositive: If a program is easy to use, it should be easy to write. Therefore, I have avoided using C to program for Windows. Instead, I use Actor.

The Whitewater Group, mindful of Actor's emergence as a serious Windows development language, has decided to provide two new programming tools for Actor: WinTrieve, an ISAM (Indexed Sequential Access Method) for Actor (and C) programs running under Windows; and the Whitewater Resource Toolkit, an editor for building and editing all the miscellaneous resources (menus, accelerators, cursors, and so on) needed by a Windows program.

Before these tools were around, developing comparable programs for Windows meant writing these programs in C. In this article, I'll examine these tools and how they're used to build an application that actually uses "real" files and Windows resources — but with considerably less suffering.

Actor 3.0

Although Actor 3.0's support of Windows 3.0 was what made the news, there were other improvements over Version 2.0 as well. In Actor 2.0, for instance, an innovative "memory swapping" technique was implemented (see the article "Object Swapping" by Jan Bottorff and Jim Bolland, *DDJ*, May 1990) whereby the 640-Kbyte barrier was effectively broken, allowing you to write extremely large Actor programs. With Actor 3.0, this technique has been extended to use Windows' standard and enhanced 80386 modes to create applications up to 2 Mbyte in size.

Another feature worth mentioning is a streamlined seal-off procedure. The term "seal-off" refers to the process of removing the development classes from the workspace so just the application objects and classes remain: In effect, "compiling" the Actor program so it can run as a standalone Windows application. In prior versions of Actor, this could be cumbersome. In Versions 2.0 and 3.0, there's a menu option and a dialog to fill out, and the rest of the seal-off process is automatic.

There are also numerous other improvements in the product, including a *System* object for encapsulating operating system dependencies, and more Windows-specific classes. The new Windows classes include convenient Windows printing and dynamic menus and dialogs. The class library improvements have been added with an eye toward compatibility with prior versions of Actor, a lesson Borland and Microsoft could learn from the Whitewater Group.

Dynamic dialog and menu classes make rapid application prototyping much easier. In earlier versions of Actor, there were only simple modal and nonmodal dialogs available as classes without resorting to the Actor's Windows Call facility. This new version allows complex dialogs to be built with more object-oriented means, as shown in Example 1.

In the *recDlg( )* method shown in Example 1, a new dialog is constructed and returned. The *setSize( )* message gives the overall dimensions of the dialog box, while each *addItem( )* adds a control or field. Static text, edited fields, and a pushbutton to cancel the dialog are added, along with their control IDs. Dynamic Dialogs are used in the sample program to create a multi-field input dialog for the record being added or updated in the database.

Other Actor 3.0 features include be-

```
/* Build a record dialog. */
Def recDlg(self, dPhone, dPerson, dCompany | D)
{ D := new(DialogDesign);
  setSize(D, 8@8, 185@120);
  addItem(D, newStatic(DlgItem, "Phone:", 100, 5@10, 40@10, 0));
  addItem(D, newEdit(DlgItem, dPhone, PHONE, 50@10, 35@12, 0));
  addItem(D, newStatic(DlgItem, "Person:", 100, 5@25, 40@10, 0));
  addItem(D, newEdit(DlgItem, dPerson, PERSON, 50@25, 125@12, 0));
  addItem(D, newStatic(DlgItem, "Company:", 100, 5@40, 40@10, 0));
  addItem(D, newEdit(DlgItem, dCompany, dCompany, 50@40, 35@12, 0));
  addItem(D, newButton(DlgItem, "Cancel", IDCANCEL, 115@95, 40@14, 0));
  ^D;
}
```

Example 1: *Constructing and returning new dialog*

Marty is a software engineer with Allen Testproducts Inc. of Kalamazoo, Michigan. He is the author of Object-Oriented Programming Featuring Actor, *published by Scott Foresman & Co.*

(continued from page 86)
ing able to more easily go between Actor and C or Assembler, including writing your own primitives in C and linking them into the Actor image.

WinTrieve

WinTrieve is a data manager for both Actor and C that provides an Indexed Sequential Access Method (ISAM) for Windows programs. Without it, fledgling Windows programmers must write their application using only the file access provided through the basic Actor or C language. This means there's no support for addition and retrieval of keyed records. Since many (if not most) business-type applications require this type of function, it meant hand-building this support into Actor or C programs — not the most entertaining job.

Actor can use Windows' standard and enhanced 80386 modes to create applications up to 2 Mbyte in size

WinTrieve provides the capability to create, add, delete, and retrieve records using a key. While not a complete database-management system, this support goes a long way to building a DBMS and in fact provides a sizable chunk of the function needed by most business programs. WinTrieve also provides some limited relational capabilities, such as linking fields between ISAM files. Locking is available in a fully automatic mode, a manual mode, or none. Both entire files and individual records may be locked.

WinTrieve has three parts: An ISAM server, which runs as a separate Windows application, and two Application Program Interfaces (APIs), one for Actor, the other for C. The Actor API consists of several classes that are loaded into the Actor image. These include an *IsamManager* class, an *IsamFile* class, and several *IsamKey* classes. The C API consists of header files and libraries (one for each memory model) and can be linked into your Microsoft C programs.

The ISAM Manager is loaded as a separate Windows program and runs "outside" of the Actor workspace. This means that programs written outside of Actor can use it, provided they adhere to the protocol. This has been

documented in the WinTrieve manual and shouldn't prove any problem to experienced Windows programmers.

Using an object-oriented language has advantages when writing programs that use an ISAM. The OOP metaphor means you can think about the ISAM file as another collection or file object. Once written, the often-convoluted logic of dealing with recovery, transaction locking, and journaling can be safely placed in the class tree and inherited without a lot of fuss later.

WinTrieve is an access method, providing the application with services to build and use indexed files; it's not a complete DBMS. This limitation is both good news and bad. The downside is that you don't get a lot of the features of a full DBMS — query facilities and application and report generators. The good news is that you get a simpler, more flexible tool with considerably less overhead. If you need the additional DBMS-like facilities, you can build them in Actor or C.

Whitewater Resource Toolkit

One of the limitations in earlier versions of Actor was that programmers were dependent on the Microsoft Windows SDK for customizing application resources. What a drag. The power of a state-of-the-art language such as Actor was harnessed in tandem with a plowhorse like the Resource Compiler. It made using static resources in Actor programs a chore, enough so — unless you were writing production-quality programs in Actor — that you didn't do it.

The Whitewater Resource Toolkit is a solution to this problem. It supports both C and Actor programs and can handle resources in RES, BMP, ICO, CUR, and EXE formats. The individual editors in it work like MacDraw or MacPaint: You are presented with a graphical view of the resource you are editing and you use a selection of tools, chosen from a palette, to edit them.

At the time I'm writing this, the Resource Toolkit doesn't run support for Windows 3.0. But, by the time you're reading this, the Resource Toolkit will likely support Windows 3.0. Until then, you'll have to edit resources under Windows 2.1, and bring them over.

Within the Resource Toolkit are editors for each type of resource: graphics editors (for bitmaps, cursors, and icons), a menu editor, a dialog editor, an accelerator editor, and a string editor. These editors are all linked by a tool called the "Mover," which allows you to view, move, copy, select, and delete various resources in various files.

The editors are all organized along the same lines. A File menu loads and saves the resource you're editing, and an Edit menu cuts, copies, and pastes parts of a shape. In the graphics editors, the palettes have tools for drawing lines, rectangles, and ellipses, and moving shapes and changing their colors.

The Resource Toolkit is easy and fun to use and makes you want to customize your application beyond the limits of good taste and user-friendliness. It's also entirely written in Actor and makes a convincing demonstration of what the language is capable of.

The term "seal-off" refers to the process of removing the development classes from the workspace so just the application objects and classes remain

Putting it to Work

To demonstrate and evaluate these new tools, I took the Book Browser application provided with WinTrieve and modified it into a Rolodex file. ROLO.H (Listing One, page 132) is the application header file. As in a C program, this file contains constants, including menu item numbers and dialog constants.

ROLO.ICO (available electronically, see "Source Code Availability" on page 3) is the Rolodex application icon. I took the icon provided for the Book Browser and modified it using the Resource Toolkit, adding a spiral binding to the side of the "book." Voilá! instant Rolodex.

ROLO.ACT (Listing Two, page 132) adds some additional methods to the *String* class for the Rolodex. ROLO.RC (see "Availability") is the resource file for the sealed-off application. To build this file, I simply modified what was in the Book Browser's resource file using Actor's File Editor.

ROLOAPP.CLS (Listing Three, page 132), ROLOFILE.CLS (Listing Four, page 132), and ROLOWIND.CLS (Listing Five, page 132) are all class files for the application. Listing Three implements the RoloApp class, the "wrapper" for the sealed-off application. It has a sin-

gle message, *init( )*, sent when the application is started. It performs necessary initialization and passes control to *RoloWindow*, the application's main class. One of the initialization steps performed in *init( )* is to start the ISAM Manager, and either create or open the Rolodex database.

ROLOFILE (Listing Four) is a descendant of the *IsamFile* class. It defines the structure of the ISAM file that contains the Rolodex. The ISAM file has three fields: The entry's phone number, the primary key, the person's name, a secondary key, and the person's company affiliation. All these fields are 30-character strings. The sole purpose of this class is to pass this initialization information to the ISAM Manager when the application is started.

Finally, ROLOWIND (Listing Five) implements the workhorse of the application; the *RoloWindow* class. Since we're developing a standalone Windows program, it'll also be the window that gets control when the program starts. The *RoloWindow* class is a descendant of *TextWindow*, so it automatically inherits all of that class's features, including text entry and a text cursor. But the *RoloWindow* class adds a menu handler that performs the functions of the application, including perusing the database by moving forwards and backwards and inserting and deleting entries.

The *RoloWindow* class is relatively large because it must support an elaborate protocol for Windows. This includes not only the messages needed to handle menu commands to the window, but also each menu function's message, such as *deleteRec( )* to remove the current record from the file and *updateRec( )* to update the current record if it's been changed.

The bulk of the work in RoloWind.CLS is done in *validateInput( )* and its private messages *getPerson( )* and *getPhone( )*. These obtain and validate input from the user using Actor's Dynamic Dialogs. The *recDlg( )* message constructs this dialog when the *RoloWindow* is initialized using several *addItem( )* messages. It builds a Dictionary containing the name of each field as a key symbol and the field's contents as the value. These are not to be confused with the actual ISAM records, which have the same field names but are in all uppercase.

The amount of code in *RoloWindow* is reduced by using Actor's powerful *perform( )* message. This allows us to create a collection of methods and messages for all the possible responses to RoloWindow's *command( )* message. When the *command( )* message is sent with a menu number after a menu item is picked by the user, *command( )* simply passes control to the method matching the menu ID. No multiway branches are needed. This means that new menu options can be easily added to the Rolodex later.

To get the application working, you must first load the ISAM Manager's classes into the Actor workspace. On my home PC (a 10-MHz 286 with only 640 Kbyte of RAM) I had to chop Actor's dynamic and static storage allocations before both the ISAM Manager and Actor could coexist under Windows. The Whitewater Group recommends having at least 1 Mbyte of RAM when using WinTrieve and Actor together. On my work PC, a 20-MHz 386 with 2 Mbyte of RAM, this was not a problem. The sealed-off application, however, ran fine on both machines.

Once the ISAM manager is loaded, you need to load the Rolodex classes by loading the ROLO.LOD (see "Availability") load file. You can then start the Rolodex interactively using:

```
Sam := new(RoloApp);
init(Sam, nil);
```

Conclusion

Actor was always a convenient programming tool for Windows, and a good way to become familiar with Windows' facilities even if you wrote the final program in C. With Version 3.0, Actor is now better suited to developing real applications. The addition of WinTrieve and the Whitewater Resource Toolkit boost Actor's capabilities as a professional applications and rapid prototyping language. Programmers despairing of C for writing Windows programs are encouraged to consider Actor for their next programming project. You might find that programs that are easy to use can also be easy to write.

DDJ

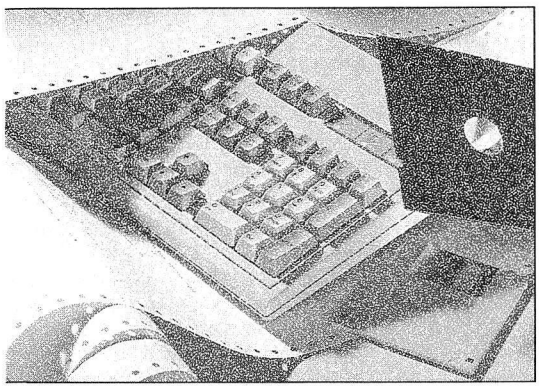
Products Mentioned

Actor 3.0 $695
Whitewater Resource Toolkit 1.0 $195
WinTrieve 1.0 $395

The Whitewater Group
1800 Ridge Ave.
Evanston, IL 60201-3621
800-869-1144

(Listings begin on page 132.)

Vote for your favorite feature/article.
Circle Reader Service **No. 7.**



Windows 3.0 Application Development

A single tool just won't cut it anymore

Walter Knowles

For users, Windows 3.0 has significantly increased the visibility of Graphical User Interfaces (GUIs). For programmers, however, creating sophisticated Windows applications has proven to be no mean feat. Even the simplest Windows programs require the overhead of message reception and dispatch functions, and demand that programmers master new interfaces. To go beyond the "simple" dialog-box/built-in control level of interface, programmers must often write their own code for manipulating text, identifying the location clicked on by the mouse, and displaying graphics.

Furthermore, Windows, and indeed any GUI, pays a penalty in speed and program size for its user friendliness, and programmers usually need to compensate by making data storage as compact and efficient as possible. This was certainly the case with the checking account manager which I describe in this article. This application provides cash-based accounting for an individual or small business by letting the user balance multiple checkbooks and distribute transactions across multiple categories of expenses.

This checking account manager consists of setup screens — for accounting information, names and addresses of check recipients, recurring entries, and so on. The part of this application that I'll focus on is a general check entry screen that uses data from these setup screens (see Figure 1). Key to this user interface are the combobox-style selectors which let the user select transaction types, check payees, and distribution accounts. These comboboxes allow the user direct access to the database, based on either the name or code order. Another very significant element is the visual representation of the checkbook register that lets the user actively select which check to edit.

Walt is a programmer for Asymetrix Corp. and can be reached at 110 110th Avenue N.E., Suite 717, Bellevue, WA 98004.

The Development Process

Our development team consisted of an accountant (who did the specification research), a User-Interface (UI) designer, two ToolBook programmers, and a Windows/C programmer. Although ours was a five-member team, there's no reason why one or two broadly disciplined individuals couldn't have accomplished what we did.

ToolBook was the front-end for the application because of its graphical richness, programming language, and relatively straightforward interface to Windows Dynamic Link Libraries (DLLs). For the database engine, we chose db_VISTA III Windows DLL because of its relatively small size and significant speed advantage over the more traditional relational or flat-file manager. For optimizing our ToolBook/db_VISTA application, we used the Windows SDK and Microsoft's C 5.1 compiler. We also used the SDK reference material to guide our interface to the Windows kernel functions for interfacing ToolBook and db_VISTA.

After we had defined the basic functionality for the program, the accountant sketched the screens using ToolBook's drawing tools. Because our main "user" developed the basic components of the UI, we avoided the usual cycles of capturing the user's requirements and refining the idea.

Because ToolBook provides a functional flat-file database with its record field object, we took the UI designer's screens, put some minimal functionality behind them, and had a functional application prototype up and running in slightly more time than it would have taken using a slide show demo program. We then tried out the program on "real" users to test our general conceptions and data flows.

While the UI designer was busy polishing the screens and menus, the programming team focused on implementing the database in its final form. ToolBook's internal object-oriented database was ideal for the prototype, but, like all OODBMSs, the overhead of using ToolBook's database for a traditional "data processing" database appeared to be too high.

As part of its OpenScript language, ToolBook provides an interface that allows links to external database engines which can store data outside the ToolBook OODBMS. (Additionally, the interface allows manipulation of the Windows functions, including Windows memory management.) To use this interface, the external functions must be

(continued from page 92)
in a Windows-compatible DLL and must not need to call back into the instance that calls them. This task is made significantly easier if the external functions use the Pascal calling convention (as is typical of Windows functions).

In reality, a checking account manager is a hierarchically-structured complex accounting system. Although this structure is more clearly modeled with a hierarchical database than a relational one, we needed more flexibility than a traditional hierarchical database provides. We also needed some relational features (keys, for example), and some common features of small, fast datasets with multiple owners. And, given our time constraints (we had to prove the technology *and* deliver the application to the client on schedule) we needed a working Windows 3.0-compatible DLL. The db_VISTA database engine provided this.

Building the Database

Our basic database design goal was to make the performance fast. To compensate for the overhead display of Windows and ToolBook with their complex graphics (for example, ToolBook pages take up more disk space than db_VISTA records, and disk access is understandably slower), we knew we could optimize data access so that the users would feel the system was responsive. Because the fastest way to access data in a hierarchical database is through the set pointers, we defined a database that has more small sets than might be usual for an application of this size.

Ordered sets, because they are doubly-linked lists, degrade linearly with the number of entries in the set. We thus designed the database so that the typical maximum number of members of an ordered set would be 150, and the average number of members would stay below 50. This necessitated making some of the sets "sets-of-sets."

We then used keys for access across multiple sets. Because keys are used primarily for reporting rather than for transaction processing, we were willing to have slightly slower access. The end result was the design shown in Figure 2. We then coded this database in db_VISTA's data definition language. (See Listing One, page 136.) The first part of the definition defines three database files and a key file. db_VISTA could have combined all records into one file, but to minimize disk usage, we chose to separate them. The second section of the definition defines the record structures in a rather C-like language; the third defines the sets we are using. We used two orderings of sets:

- *Ascending*, which orders the set from lowest to highest in either the numeric or text collation sequence. This ordering can exact a penalty in insertion because it forces a linear search of the link list for the insertion point.
- *Next*, which inserts the new record into the set sequence directly after the current record. This "ordering" is very fast on the insertion, but not readily searchable.

db_VISTA provides three other orderings: first, last, and descending. Sets can be ordered (for example in *payor_accountcode*) by more than one field, as long as the fields are in the same record.

Coupling db_VISTA and ToolBook

Once the database was designed, we had to hook up db_VISTA with ToolBook. ToolBook's OpenScript language

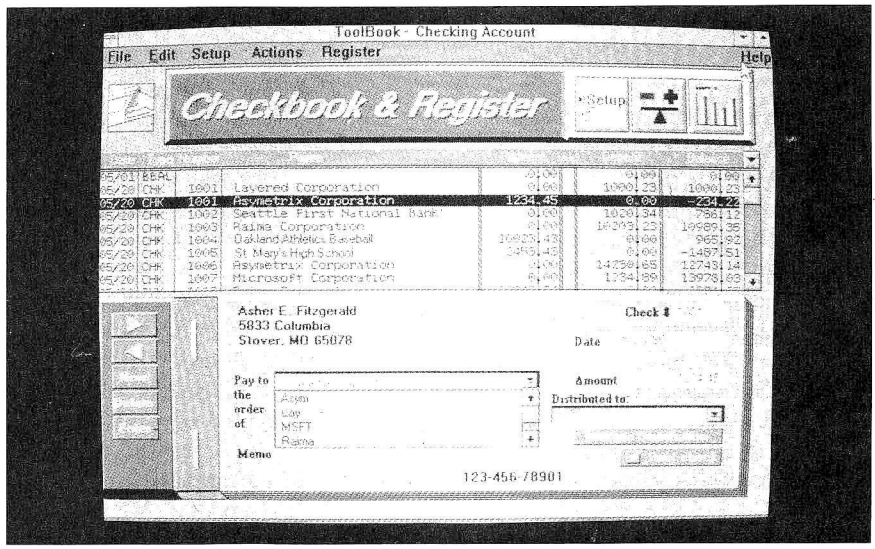


Figure 1: *Main setup screen for the checkbook accounting system*

(continued from page 94)
provides three means for linking to and using a DLL: LinkDLL, function calls, and pointers and pointer conversions.

LinkDLL identifies the functions in the DLL that ToolBook will call and specifies the DLL which contains them. Functions are then prototyped using

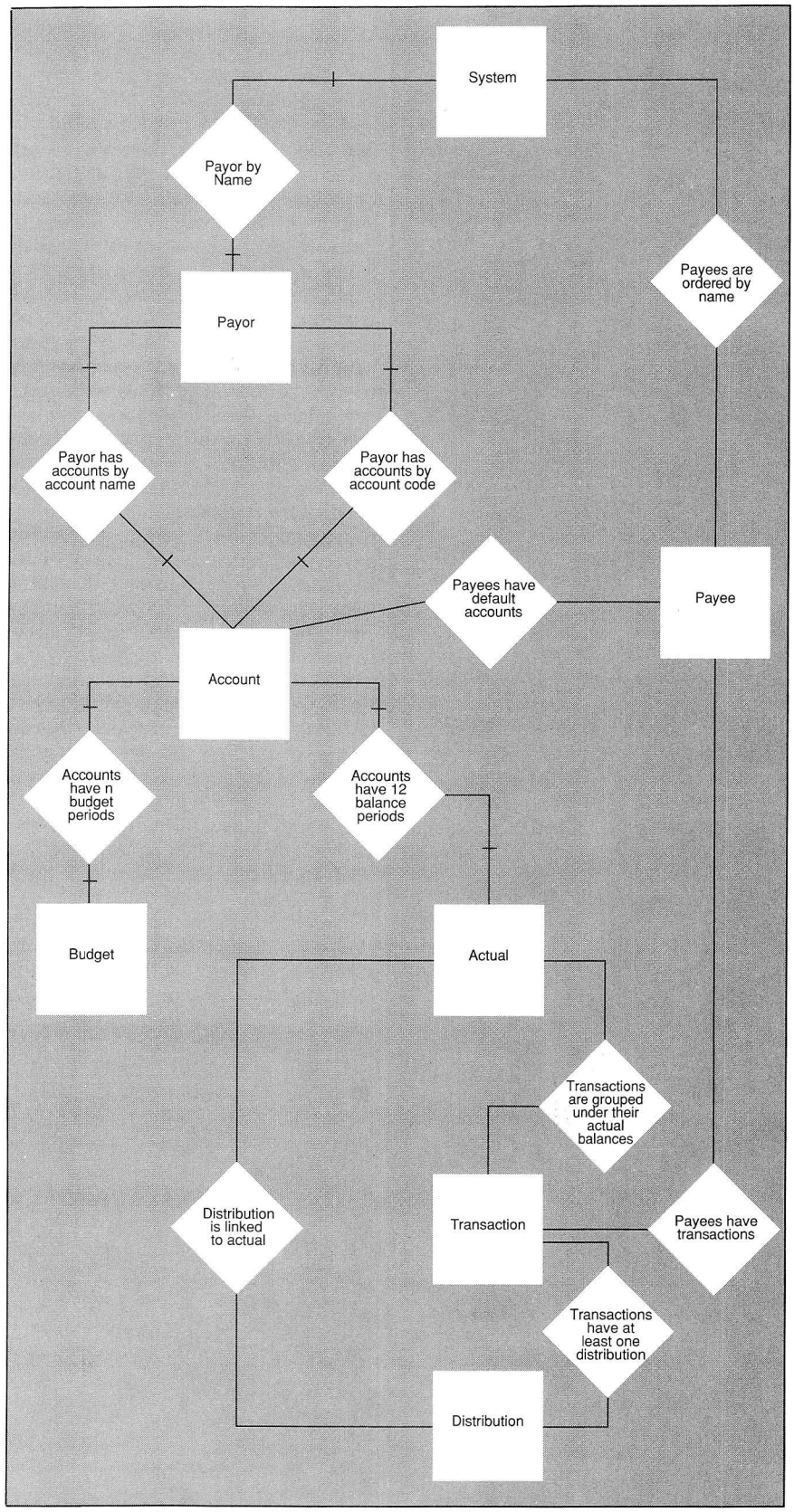


Figure 2: *Database design*

(continued from page 96)
the data types for which ToolBook has built-in coercion. (See Table 1.)

Listing Two (page 136) is the LinkDLL structure for the check manager application where we link to four DLLs: The Windows 3.0 kernel for memory management functions; ToolBook's *tbkfile.dll*, to check the drive and directory; db_VISTA's *vista.dll* for database management (we use about half the functions of the DLL in this application); and *chekmate.dll*, our own helper.DLL.

Once a function has been linked in a LinkDLL structure, the function is called just like an OpenScript intrinsic function. If we call db_VISTA's *find first member of set* function in C, we use something like *error = dt_findfm (PAYEE_ACCOUNT, hCurrentTask, nDBnum);* In ToolBook, it's much the same: *set vError to dt_findfm(20002, svhCurrentTask, svDBnum).*

Most DLLs pass data back and forth by reference as well as by value; db_VISTA is no exception. OpenScript provides a pointer dereferencing function for each of its coercible data types. For example, the function for converting to and from a LONG is *pointerLONG.* To dereference the second element of a db_VISTA database address array (they are longs declared as *DBA payee[20];*), we would set the second element to *pointerLong(4,payee)* and the third to *get pointerLong(8,payee-,39421).*

BYTE	unsigned char
INT	signed int
WORD	unsigned int
LONG	signed long
DWORD	unsigned long
FLOAT	float
DOUBLE	double
POINTER	void far *.
STRING	LPSTR, or void far **.

*ToolBook displays a pointer as segment,offset
**ToolBook requires a string to be zero terminated.

Table 1: *DLL functions called by ToolBook*

Because ToolBook has its own internal values system, you can't directly pass by reference. When your DLL requires pointers instead of values (and db_VISTA does), you need to allocate Windows global memory that both ToolBook and your DLL can use.

Memory Management Calls

ToolBook is "just" a Windows program, and db_VISTA is "just" a Windows DLL. That means that all memory has to be manipulated through Windows' *kernel.EXE*'s memory management functions. The five key functions are listed in Table 2.

Because all of these calls can fail under low-memory conditions, check that your calls were executed and recover if not. We haven't included this error checking in the examples, but Listing Three (page 138) shows how it could be done. *InitGlobals* allocates and locks the three global memory blocks we use as sharable, zero initialized memory. *FreeGlobals* does the cleanup. It is called when the system shuts down. The ToolBook system variables, which point to the Windows globals, are set to null so that the ToolBook coercion system will trap uninitialized memory pointers for us, rather than sending them off to Windows to cause general-protection violations.

Global Memory Blocks and Communication with the DLL

Because ToolBook's values are not directly available to the DLL world, we constructed two primary global blocks. The first is a simple 255-byte buffer that we can pass back and forth between ToolBook and db_VISTA. You'll see it in the listings as *svptrDbBuffer.* Since ToolBook is a protected-mode only application, leaving the memory locked makes no difference to Windows use.

The other main block is a structure which is mainly an array of handles to Windows memory. To maximize performance, we needed to buffer a num-

Function	Description
GlobalAlloc	Returns a handle to memory. Handles are put in a table, so that memory can be moved around unless locked down. You can't directly use the allocated memory unless you "lock it down" and get a pointer to it.
GlobalReAlloc	Resizes a memory block. This function returns a new handle which may be the same as the old one.
GlobalLock	Takes a handle and returns a pointer to fixed memory. Because Windows is multitasking, you don't want to lock memory for any longer than absolutely necessary.
GlobalUnLock	Unlocks the memory so that it can move. Because the memory can move, this function invalidates the pointers to the block.
Global Free	Frees the memory so that it can be reused.

Table 2: *Memory management functions*

ber of relatively small blocks of data, particularly the combobox selectors for accounts and vendors. The structure definition for this control block is in Listing Four (page 138). The code and name array handles point to these selectors and the database addresses for each of these selector fields.

We could access this block by using the pointer function when we need it, but it's better to centralize *set*ting and *get*ting values, particularly when the structure is changing. Listing Five (page 138) is the accessor code for the ToolBook side of this control block. The first *to get* handler, or ToolBook function *to get hControl*, returns the value of the individual element by getting the offset through the *ControlOffset* function and dereferencing the structure. The other handler in this example is orthogonal to the *get* function *to set hControl* and uses the same *ControlOffset* function to get the offset before using the second form of the pointer functions to set values in the control block.

Shell Calls to db_VISTA

The db_VISTA engine, which was designed to be called from a C program, provides several header files to make the interface in a multiple-tasking environment such as Windows simple and clean. db_VISTA requires the calling program to specify both a pointer to an instance control block and the database identifier. In order to emulate the simplicity, we began to shell calls to db_VISTA so that we only needed to pass the relevant parameters inside our main program. We also used *to set* and *to get* handlers to provide parallelism in the function structure. Because of space constraints, the actual shell code for general-database access is not included with this article but is available electronically (see "Source Code Availability" on page 3).

Optimizing

While ToolBook, like any other general-purpose programming environment does many things well, there are areas that, either for speed or ease of maintenance, require optimization. We identified the key areas, then applied a good dose of "vitamin C." The most important step in any optimization is to identify what really needs to be optimized. We identified three key areas where optimization would give us noticeable performance benefits:

- Putting the text and database addresses of the combobox selectors in global memory
- Building and maintaining the check register image in global memory
- Maintaining account balances

The combobox selectors are on the check-image page and on various other pages. We could have left them there, but they need to be updated throughout the system, and this takes time. We elected to keep these in global memory and copy them into ToolBook fields when needed.

By keeping the database addresses available, we can access the account and payee information quickly; we can also connect records quickly without having to actually retrieve them from the database. It is significantly faster to dereference the 25th element of an array than to get the 25th item of a ToolBook list. Therefore, we put this information in global memory. We first implemented this optimization using ToolBook's OpenScript. (For those of you who are interested, this code is also available online and on disk.)

After we completed all optimizations, we came back and optimized our ToolBook optimization in MS C 5.1 as shown in Listing Seven on page 139. (The original ToolBook version appears in Listing Six, page 138.) This code demonstrates one of the best uses of ToolBook as a prototyping tool. Except for breaking *CreateSelectorList* into two functions, there is a direct correspondence in the C code and the OpenScript code. By prototyping even those functions, which we knew would be recoded in C, we saved several compile-link-test cycles in ToolBook's interactive environment.

We had planned from the start to manage the register and its balances in a helper.DLL. While computation in ToolBook is no slouch, we could get a substantial increase in speed by using integer arithmetic and array manipulation in the helper.DLL rather than ToolBook's floating-point arithmetic and string manipulation.

There are three columns in a check register: deposits, withdrawals, and balances. We stored this as a 3-by-*n* array of longs (longs give us +/– $21 million, definitely large enough for most people's checkbooks!). Because the balances are never stored in the database, we calculate them on-the-fly from the *registerNumbers* array. (You can get the balance maintenance code online or on disk.)

We planned to maintain the check register image and its database address array in global memory as part of the initial design. The check register, as you can see in Figure 1, is an array of the check number, date, description, whether or not the check has been reconciled, the deposit, withdrawal, and balance amounts. We build each line for a given month in the helper.DLL and once all of the lines are created, we copy the image into a ToolBook field. An important part of this optimization is the same technique which we used with the combobox text: We maintain an array of database addresses which corresponds to the register. This allows us to directly access a check by its database address. (The code for building the register is also available online.)

Product Information

ToolBook v. 1.0
Asymetrix Corp.
110 110th Ave. N.E., Ste. 717
Bellevue, WA 98004
206-462-0501

db_VISTA III v. 3.15
Raima Corp.
3245 146th Pl. S.E.
Bellevue, WA 98007
206-747-5570

Windows 3.0
Windows 3.0 SDK
Microsoft C 5.1
Microsoft Corp.
One Microsoft Way
Redmond, WA 98052-6399
800-227-4679

Conclusion

By using development tools where they have their greatest strengths — ToolBook for implementing a graphically sophisticated interface, db_VISTA for handling raw data, and C for optimization — programmers can bring an application from conception through implementation in much less time than is required for typical GUI applications. Even with the overhead of a large development and delivery environment such as ToolBook, we were able to provide solid performance by carefully choosing where to optimize and gaining the greatest advantage from our optimization effort.

Acknowledgments

The C code examples to this article are the effort of the Windows developer on our team, Leslie Gardenswartz, and some of the best aspects of the checkbook manager's internal design are due to his efforts.

DDJ

(Listings begin on page 136.)

Vote for your favorite feature/article.
Circle Reader Service **No. 8.**

Listing One *(Text begins on page 16.)*

```
/* File : OOP.PRO -- Include file that adds inheritance mechanism
   and message passing facility. This file also declares the
   object-oriented predicates msg(), method(), has(), and is_a().
   Objects are implemented using a technique known as frames;
   the inheritance mechanism is based on the article "Suitable for
   Framing" by Michael Floyd (Turbo Technix, April/May '87).
   Michael Floyd -- DDJ -- 8/28/90 -- CIS: [76703,4057]  or MCI Mail: MFLOYD
*/

DOMAINS
   object    = object(string,slots)          % not actually used in examples
   objects   = object*

   slot     = slot(string,value)
   slots    = slot*

   value    = int(integer) ; ints(integers) ;
              real_(real) ; reals(reals) ;
              str(string) ; strs(strings) ;
              object(string,slots,parents) ; objects(objects)
   parents  = string*
   integers = integer*
   reals    = real*
   strings  = string*

% --- OOP preds to be used by the programmer ---
DATABASE
   has(string,slots)                    % storage for instance vars
   is_a(string,string)                  % hierarchy relationships
PREDICATES
   msg(string,string)                   % send a message to an object
   method(string, string)               % define a method

% --- Internal Predicates not called ditrectly by the programmer ---
   inherit(string,slot)                 % inheritance mechanism
   description(string,slot)             % search for matching clauses
   member(slot,slots)                   % look for object in a list
CLAUSES

/* Inheritance Mechanism */
   inherit(Object,Value):-
      description(Object,Value),!.
   inherit(Object,Value):-
      is_a(Object,Object1),
      inherit(Object1,Value),
      description(Object1,_).
   description(Object,Value):-
      has(Object,Description),
      member(Value,Description).
   description(Object,slot(method,str(Value))):-
      method(Object,Value).

/* Simple message processor */
   msg(Object,Message):-
     inherit(Object,slot(method,str(Message))).

/* Support Clauses  */
    member(X,[X|_]):-!.                 % Find specified member in a list
    member(X,[_|L]):-member(X,L).
```

End Listing One

Listing Two

```
/* File: FIGURES.PRO -- Object Prolog example that models FIGURES example in
   Turbo C++ and Turbo Pascal documentation
   Michael Floyd -- DDJ -- 8/28/90
*/

include "bgi.pro"
include "OOP.PRO"

domains
   key = escape; up_arrow; down_arrow; left_arrow; right_arrow; other
database - SHAPES
   anyShape(string)

PREDICATES                                  % Support predicates
   horiz(integer, integer, string)
   vert(integer, integer, integer)
   readkey(integer, integer, integer)
   key_code(key, integer, integer, integer)
   key_code2(key, integer, integer, integer)
   repeat
   main

CLAUSES
/* Methods */

/* point is an example of an Abstract object. Note that variables passed
   through the database must be explicitly called by the child
   method (i.e. variables are not inherited). */

   method(point, init):-
      assert(has(point,[slot(x_coord,int(150)),
                        slot(y_coord,int(150))])).
   method(point, done):-
      retractall(has(point,_)).
   method(point,show):-!,
      has(point,[slot(x_coord,int(X)),
                 slot(y_coord,int(Y))]),
      putpixel(X,Y,blue).
   method(point,hide):-!,
      has(point,[slot(x_coord,int(X)),
                 slot(y_coord,int(Y))]),
      putpixel(X,Y,black).
```

```
/* Example of a virtual method */
   method(point,moveTo):-!,
      anyShape(Object),
      msg(Object,hide),
      retract(has(point,[slot(x_coord,int(DeltaX)),
                         slot(y_coord,int(DeltaY))])),
      msg(Object,show).
   method(point,drag):-
      has(point,[slot(x_coord,int(X)),
                 slot(y_coord,int(Y))]),
      anyShape(Shape),!,
      msg(Shape,show),
      repeat,
         readkey(Key, DeltaX, DeltaY),
         assertz(has(point,[slot(x_coord,int(DeltaX)),
                            slot(y_coord,int(DeltaY))])),
         msg(point,moveTo),
      Key = 27.

 /* Circle Methods */
   method(circle, init):-!,
      method(point, init),
      assert(anyShape(circle)).
   method(circle, done):-!,
      retract(anyShape(circle)),
      method(point, done).
   method(circle, show):-!,
      has(point,[slot(x_coord,int(X)),
                 slot(y_coord,int(Y))]),
      setcolor(white),
      circle(X,Y,50).
   method(circle, hide):-!,
      has(point,[slot(x_coord,int(X)),
                 slot(y_coord,int(Y))]),
      setcolor(black),
      circle(X,Y,50).
   method(circle, drag):-
      msg(circle, hide),
      is_a(circle, Ancestor),
      msg(Ancestor, drag),
      msg(circle, show).

/* arc Methods */
   method(arc, init):-!,
      assert(anyShape(arc)),
      assert(has(point,[slot(x_coord,int(150)),
                 slot(y_coord,int(150))])),
      assert(has(arc,[slot(radius,int(50)),
                      slot(startAngle,int(25)),
                      slot(endAngle,int(90))])).
   method(arc, done):-
      retract(anyShape(arc)),!,
      retractall(has(arc,_)),
      msg(point, done).
   method(arc, show):-
      has(point,[slot(x_coord,int(X)),
                 slot(y_coord,int(Y))]),
      has(arc,[slot(radius,int(Radius)),
                      slot(startAngle,int(Start)),
                      slot(endAngle,int(End))]),!,
      setcolor(white),
      arc(X, Y, Start, End, Radius).
   method(arc, hide):-
      has(point,[slot(x_coord,int(X)),
                 slot(y_coord,int(Y))]),
      has(arc,[slot(radius,int(Radius)),
                      slot(startAngle,int(Start)),
                      slot(endAngle,int(End))]),!,
      setcolor(black),
      arc(X, Y, Start, End, Radius).
   method(arc,drag):-
      msg(arc, hide),
      is_a(arc,Ancestor),!,
      msg(Ancestor,drag),
      msg(arc, show).

/* rectangle Methods */
   method(rectangle,init):-
      has(rectangle,[slot(length,int(L)),
                     slot(width,int(W))]),!.
   method(rectangle,init):-!,
      write("Enter Length of rectangle: "),
      readint(L),nl,
      write("Enter Width of rectangle: "),
      readint(W),nl,
      assert(has(rectangle,[slot(length,int(L)),
                     slot(width,int(W))])).
   method(rectangle,done):-
      retract(has(rectangle,[slot(length,int(L)),
                     slot(width,int(W))])),!.
   method(rectangle,draw):-!,
      has(rectangle,[slot(length,int(L)),
                     slot(width,int(W))]),
      write("Z"),
      horiz(1,L,"D"),
      write("?"),nl,
      vert(1,W,L),
      write("@"),
      horiz(1,L,"D"),
      write("Y").
   method(rectangle,draw):-
      write("Cannot draw rectangle"),nl.

/* Support Methods */
  horiz(I,L,Chr):-
     I <= L,!,
     TempI = I + 1,
     write(Chr),
     horiz(TempI,L,Chr).
  horiz(I,L,Chr):-!.
```

(continued on page 104)

Listing Two *(Listing continued, text begins on page 16.)*

```
  vert(I,W,L):-
     I <= W,!,
     TempI = I + 1,
     write("3"),
     horiz(1,L," "),
     write("3"),nl,
     vert(TempI,W,L).
  vert(I,W,L):-!.

/* Ancestor/Child relationships - should be stored in consult() file */
   is_a(circle,point).
   is_a(arc,point).
   is_a(triangle,shape).
   is_a(rectangle,shape).
   is_a(solid_rectangle,rectangle).

/* Generic clause to read cursor keys - used by the Drag method */
   readkey(Val, NewX, NewY) :-
      readchar(T),
      char_int(T, Val),
      key_code(Key, Val, NewX, NewY).
   key_code(escape, 27, 0, 0) :- !.
   key_code(Key, 0, NewX, NewY) :- !,
      readchar(T),
      char_int(T, Val),
      key_code2(Key, Val, NewX, NewY).
   key_code2(up_arrow, 72, NewX, NewY) :- !,
      has(point,[slot(x_coord,int(X)),
                 slot(y_coord,int(Y))]),
            NewX = X,
            NewY = Y - 5.
   key_code2(left_arrow, 75, NewX, NewY):- !,
      has(point,[slot(x_coord,int(X)),
                 slot(y_coord,int(Y))]),
            NewX = X - 5,
            NewY = Y.
   key_code2(right_arrow, 77, NewX, NewY) :- !,
      has(point,[slot(x_coord,int(X)),
                 slot(y_coord,int(Y))]),
            NewX = X + 5,
            NewY = Y.
   key_code2(down_arrow, 80, NewX, NewY) :- !,
      has(point,[slot(x_coord,int(X)),
                 slot(y_coord,int(Y))]),
            NewX = X,
            NewY = Y + 5.

   key_code2(other, _,0,0).

/* Supports the repeat/fail loop */
   repeat.
   repeat:- repeat.
```

```
main:-
  nl,
  initialize,                          % init BGI graphics
  makewindow(1,7,0,"",0,0,25,80),
  msg(circle, init),                   % create and manipulate a circle
  msg(circle, show),
  msg(circle, drag),
  msg(circle, done),
  clearwindow,
  msg(arc, init),                      % create and manipulate an arc
  msg(arc, show),
  msg(arc, drag),
  msg(arc, done),
  closegraph,                          % return to text mode

  makewindow(2,2,3,"",0,0,25,80),
  msg(rectangle, init),                % create a rectangle in text mode
  msg(rectangle, draw),
  msg(rectangle, done).

goal
   main.
```

End Listing Two

Listing Three

```
/* File: BGI.PRO -- Minimum required to detect graphics hardware and initialize
   system in graphics mode using BGI. BGI.PRE is included with PDC Prolog.
   Michael Floyd -- DDJ -- 8/28/90
*/

include "D:\\prolog\\include\\BGI.PRE"

CONSTANTS
  bgi_Path = "D:\\prolog\\bgi"

PREDICATES
  Initialize

CLAUSES
  Initialize:-
     DetectGraph(G_Driver, G_Mode),
     InitGraph(G_Driver,G_Mode, _, _, bgi_Path),!.
```

End Listing Three

Listing Four

```
include "bgi.pro"
include "support.pro"
database - figures
   anyShape(string)

point = object
   int XCoord YCoord
   method(init) if XCoord = 150, YCoord = 150.
```

```
   method(done) if retract(has(point,_)).
   method(show) if putpixel(XCoord,YCoord,blue).
   method(hide) if putpixel(XCoord,YCoord,black).
   method(moveTo) if
      anyShape(Object),
      msg(Object, hide),
      retract(has(point,_)),
      msg(Object, show).
   method(drag) if
      anyShape(Shape),
      msg(Shape,show),
      repeat,
         readkey(Key, DeltaX, DeltaY),
         XCoord = DeltaX,
         YCoord = DeltaY,
         msg(point,moveTo),
      Key = 27.
end.

circle = object(point)
   int XCoord YCoord
   method(init) if
      XCoord = 200, YCoord = 200,
      assert(anyShape(circle)).
   method(done) if
      retract(anyShape(circle)),
      msg(point,done).
   method(show) if
      setcolor(white),
      circle(XCoord, YCoord, 50).
   method(hide) if
      setcolor(black),
      circle(XCoord, YCoord, 50).
   method(drag) if
      msg(circle,hide),
      is_a(circle, Ancestor),
      msg(Ancestor,drag),
      msg(circle,show).
end.

arc = object(point)
   int XCoord YCoord Radius StartAngle EndAngle

   method(init) if
      Radius = 50, StartAngle = 25, EndAngle = 90,
      msg(point, init).
   method(done) if
      retract(anyShape(arc)),
      retractall(has(arc,_)),
      msg(point, done).
   method(show) if
      setcolor(white),
      arc(XCoord,YCoord,StartAngle,EndAngle,Radius).
   method(hide) if
      setcolor(black),
      arc(XCoord,YCoord,StartAngle,EndAngle,Radius).
   method(drag) if
      msg(arc,hide),
      is_a(arc,Ancestor),
      msg(Ancestor,drag),
      msg(arc,show).
end.
```

End Listing Four

Listing Five

```
/* File: PARSER.PRO -- Implements parser to translate ODL to Object Prolog
   code. Top-down parser; simulates parse tree through predicate calls.
   Michael Floyd -- DDJ -- 8/28/90
*/
include "lex.pro"

DOMAINS
   file = infile; outfile; tmpfile

PREDICATES
   main
   repeat
   gen(tokl)
   scan
   scan_object(tokl,string)
   findAncestor(tokl)
   init_vars
   scan_methods(string)
   getMethEnd(string,string)
   write_includes
   generate_code
   generate_methods
   generate_ancestor(string)
   generate_vars
   fixVar(tokl)
   insert_isa(tokl)
   bindvars(tokl,tokl)
   addVarRef(tokl)
   assert_temp(strings,tokl)
   construct_has(tokl)
   empty(tokl)
   isvar(tokl,tokl)
   is_op(tok)
   value(tok, integer)
   search_ch(CHAR,STRING,INTEGER,INTEGER)
   process(string)
   read(string)
   datatype(string)
    headbody(tokl,tokl,tokl)
   search_msg(tokl,tokl,tokl)
   constVar(string,tok)
   writeSlotVars
```

(continued on page 106)

Listing Five *(Listing continued, text begins on page 16.)*

```
   write_seperator(tok1)
   write_comma
   append(tok1,tok1,tok1)

CLAUSES
   repeat.
   repeat:- repeat.

/**** Parser ****/
   scan:-
      readln(ObjectStr),
      ObjectStr <> "",
      tok1(ObjectStr,ObjList),
      scan_object(ObjList,Object),
      init_vars,
      scan_methods(Object),!,
      generate_code.
   scan:- scan.

   scan_object([H|List],S):-
      member(object,List),
      str_tok(S,H),
      assert(objectname(S)),
      findAncestor(List).
   scan_object(List,_):-
      openappend(tmpfile,"headers.$$$"),
      writedevice(tmpfile),
      gen(List),nl,
      writedevice(screen),
      closefile(tmpfile),trace(off),
      fail.

   findAncestor(List):-
      member(lpar,List),
      insert_isa(List).
   findancestor(_).

   insert_isa([H|List]):-
      str_tok(S,H),
      S <> "(",
      insert_isa(List).
   insert_isa([H1,H2|List]):-
      str_tok(Ancestor,H2),
      assert(ancestor(Ancestor)).

   init_vars:-
      readln(VarStr),
      process(VarStr).

   process(VarStr):-
      fronttoken(VarStr,Token, RestStr),
      datatype(Token),
      tok1(VarStr,VarList),!.          % tokenize/init variables
   process(VarStr):-
      assert(unread(VarStr)).

   datatype(int).                      % datatypes supported
   datatype(real).

   constVar(int,int(_)).               % convert tok to string
   constVar(real,real(_)).

   read(Str):-
      retract(unread(Str)),!.
   read(Str):-
      readln(Str).

   scan_methods(MethodId):-
      readln(FirstLn),
      getMethEnd(FirstLn,Method),
      search_ch('(',Method,0,N),       % Find lpar and insert MethodId
      N1 = N+1,                        % and add comma
      fronttoken(MComma,MethodId,","),
      frontstr(N1,Method,Str,OUT1),!,
      fronttoken(Method1,MComma,Out1),
      fronttoken(Method2,Str,Method1),
      tok1(Method2,MList),             % Now Tokenize the method
      not(member(end,MList)),          % Check for End statement
      assert(methods(MList)),          % Store method list
      scan_methods(MethodId).          % Look for more methods
   scan_methods(_):- !.

   getMethEnd(Line1,ReturnLn):-
      search_ch('.',Line1,0,N),
      N <> 0,!,
      ReturnLn = Line1.
   getMethEnd(Line1,ReturnLn):-
      readln(Line2),
      fronttoken(AppendLn,Line1,Line2),
      getMethEnd(AppendLn,ReturnLn).

/**** Entry point into the code generator ****/
   generate_code:-
      objectname(Object),
      generate_ancestor(Object),
      openappend(tmpfile,"methods.$$$"),
      writedevice(tmpfile),
      generate_methods,
      generate_vars,
      vars(VarList),!,
      openappend(tmpfile,"has.$$$"),
      writedevice(tmpfile),
      write("has(",Object,",",VarList),
      write(").",nl,
      writedevice(screen),
      closefile(tmpfile),
      retract(objectname(Object)).

   generate_vars:-
      findall(Var,var(Var),VarList),   % retrieve vars
      retractall(var(_)),              % cleanup database
      fixVar(VarList),
      findall(X,var(X),Slots),         % retrieve new vars
      retractall(var(_)),              % cleanup database
      assert(vars(Slots)).             % store vars as list of slots
   generate_vars:- !.

   fixVar([]):- !.
   fixVar([slot(UpToken,Const)|Rest]):-
      upper_lower(UpToken,Token),
      assert(var(slot(Token,Const))),
      fixVar(Rest).

   generate_ancestor(Object):-
      openappend(tmpfile,"isa.$$$"),   % open temp file for is_a
      writedevice(tmpfile),            % stdout to tmpfile
      objectname(Obj),                 % get current object id
      retract(ancestor(Parent)),       % get parent in hierarchy
      write("is_a(",Obj,",",Parent,")."), % write is_a clause
      nl,
      writedevice(screen),             % stdout to screen
      closefile(tmpfile).              % close temp file
   generate_ancestor(_):-              % always succeed
      writedevice(screen),             % stdout to screen
      closefile(tmpfile).              % close temp file

   generate_methods:-
      retract(methods(Method)),!,
      headBody(Method,Head,Body),
      bindvars(Body,NewBody),
      gen(Head),
      write(":-"), nl,
      addVarRef(NewBody),
      gen(NewBody),nl,
      generate_methods.
   generate_methods:-
      writedevice(screen),
      closefile(tmpfile).

/* Binding of variable names in for has() lookups */
   addVarRef(Body):-
      findall(Variable, var(slot(Variable,_)), VList),
      findall(X, var(slot(_,X)), XList),
      assert_temp(VList, XList),
      construct_has(Body).
   addVarRef(Body).

   assert_temp([],[]):- !.
   assert_temp([V|VList],[X|XList]):-
      constVar(Type,X),
      assert(tempvar(V)),
      assert(temptype(Type)),
      assert_temp(VList,XList).
```

(continued on page 108)

Listing Five *(Listing continued, text begins on page 16.)*

```
      construct_has(Body):-
         objectname(Object),
         write("   has(",Object,",","["),
         writeSlotVars,
         write(")"),
         write_seperator(Body),nl.

      write_seperator([]):-
         write(".").
      write_seperator( ):-
      write(",").

   writeSlotVars:-
      retract(tempVar(Var)),
      retract(temptype(Type)),
      upper_lower(Var,VarId),
      write("slot(",VarId,", ",Type,"(",Var,"))"),
      write_comma,
      writeSlotVars.
   writeSlotVars:- !,
      write("]").

   write_comma:-
      tempvar(_),
      write(",").
   write_comma:- !.

/* Append two lists */
   append([], List, List).
   append([H|List1], List2, [H|List3]):-
      append(List1, List2, List3).

   search_msg([H,H2|Body],[],Body):-
      H = msg,
      H2 = lpar.
   search_msg([H|Method], [H|Head], Body):-
      search_msg(Method,Head,Body).

  search_ch(CH,STR,N,N):-                  % Search for char in string
       frontchar(STR,CH,_),!.              % and return its position
  search_ch(CH,STR,N,N1):-
       frontchar(STR,_,S1),
       N2 = N + 1,
       search_ch(CH,S1,N2,N1).

   headbody([H|Body],[],Body):-
      str_tok("if",H).
   headBody([H|Method], [H|Head], Body):-
      headBody(Method,Head,Body).

   bindvars(Method,NewMethod):-
      is_op(Op),                           % supports any operator
      member(Op,Method),                   % defined by is_op()
      isvar(Method,[H|RestMethod]),        % locate variable in method
      bindvars(RestMethod,NewMethod).      % look for more vars
   bindvars(NewMethod,NewMethod):- !.      % return Method w/out vars

   empty([]).                              % simple test for empty list

   isvar([],[]):-!.
   isvar([id(X),H2,H3|RestMethod],RestMethod):-
      is_op(H2),
      value(H3,Value),
      objectname(ObjId),
      retract(var(slot(X,_))),!,           % add "var not decl." error here
      assert(var(slot(X,H3))).
   isvar([H|Method],NewMethod):-
      isvar(Method,NewMethod).

   is_op(equals).
   is_op(plus).

   value(int(X),X).

   gen([]):- !.
   gen([H|List]):-
      str_tok(S,H),
      write(S),
      gen(List).

   write_includes:-
      write("include \"oop.pro\""),nl.


main:-
/**** Reads file (e.g, FIGURES.ODL) specified on command line. First order of
  business is to add error handling for command line processsor. ****/
  comline(Filename),
  openread(infile, Filename),
  readdevice(infile),
  repeat,
     scan,
     eof(infile),
  readdevice(keyboard),
  openwrite(outfile,"newfig.pro"),
  writedevice(outfile),
  write_includes,nl,
  file_str("headers.$$$",Headers),
  write(Headers),nl,
  write("clauses\n"),
  file_str("methods.$$$",Methods),
  write(Methods),nl,nl,
  file_str("isa.$$$",Isa),
  write(Isa),nl,nl,
  file_str("has.$$$",Has),
  write(Has),
  writedevice(screen),
  closefile(outfile),
  closefile(infile),
  deletefile("headers.$$$"),
  deletefile("methods.$$$"),
  deletefile("isa.$$$"),
  deletefile("has.$$$"),
  write("done").

goal
  main.
```

End Listing Five

Listing Six

```
/* File: LEX.PRO -- Implements scanner which tokenizes ODL. To modify, add
   appropriate DOMAIN declarations and str_tok definitions.
   Michael Floyd -- DDJ -- 8/28/90
*/

DOMAINS
   tok = id(string);
         int(integer); real_(real) ;
         plus;         minus;
         mult;         div;
         lpar;         rpar;
         comma;        colon;
         semicolon;    period;
         object;       method;
         msg;          end;
         ancestor;     var(string);
         equals;       if_;
         slash;        bslash;
         slot(string,tok); dummy

   tok1 = tok*
   strings = string*

DATABASE
   nextTok(string)                        % Token lookahead
   objectname(string)                     % current Object ID
   vars(tok1)                             % variables list
   methods(tok1)                          % methods list
   var(tok)                               % individual var
   ancestor(string)                       % tracks Object's ancestor
   unread(string)
   tempVar(string)
   temptype(string)

PREDICATES
   tok1(string, tok1)                     % entry point into the scanner
   tokenize(string,tok1)                  % tokenize a string
   str_tok(string, tok)                   % return individual token
   member(tok, tok1)                      % verify member is in list
   scan_next(string)                      % setup lookahead stack
clauses
   str_tok("int",slot(Token,int(0))):-
       retract(nextTok(Token)),!,
       assert(var(slot(Token,int(0)))),
       str_tok("int",_).
   str_tok("int",slot(dummy,int(0))):- !,
      assert(nextTok(dummy)).
   str_tok("real",slot(Token,real_(0))):-
       retract(nextTok(Token)),!,
       assert(var(slot(Token,real_(0)))),
       str_tok("real",_).
   str_tok("real",slot(dummy,real_(0))):- !.
   str_tok("(", lpar):- !.
   str_tok(")", rpar):- !.
   str_tok("=", equals):- !.
   str_tok("+", plus):- !.
   str_tok("-", minus):- !.
   str_tok("*", mult):- !.
   str_tok("/", div):- !.
   str_tok("\"",bslash):- !.
   str_tok(",", comma):- !.
   str_tok(":", colon):- !.
   str_tok(";", semicolon):- !.
   str_tok(".", period):- !.
   str_tok("if", if_):- !.
   str_tok("object", object):- !.
   str_tok("method", method):- !.
   str_tok("msg", msg):- !.
   str_tok("end", end):-!.
/*   str_tok(Var, var(Var)):-
       frontchar(Var,X,_),
       X >= 'A', X <= 'Z'.*/
   str_tok(ID, id(ID)):-
      isname(ID),!.
   str_tok(IntStr,int(Int)):-
      str_int(Intstr,Int).

/* Entry point into the scanner */
   tok1(Str, Tok1):-
      fronttoken(Str, Token, RestStr),
      scan_next(RestStr),
      tokenize(Str,Tok1).
   tokenize("",[]):- !, retractall(nexttok(_)).
   tokenize(Str, [Tok|Tok1]):-
      fronttoken(Str, Token, RestStr),
      str_tok(Token, Tok),
      tokenize(RestStr, Tok1).
   scan_next("").
   scan_next(RestStr):-
      fronttoken(RestStr, NextToken, MoreStr),
      assert(nexttok(NextToken)),
      scan_next(MoreStr).
   member(X,[X|_]):-!.
   member(X,[_|L]):-member(X,L).
```

End Listings

Listing One *(Text begins on page 20.)*

```
{$A+,B-,D-,E-,F+,I+,L-,N-,O-,R-,S-,V+}
Unit Dict;
Interface
{ DICT.PAS dictionary object and methods to create and use a superimposed
  code dictionary.  Copyright Edwin T. Floyd, 1990. }

Type
  Dictionary = Object
    DictArray : Pointer;  { Pointer to dictionary bit array }
    DictCount : LongInt;  { Number of key entries in this dictionary }
    DictSize : Word;      { Number of bytes in dictionary bit array }
    DictBits : Byte;      { Number of bits per key entry }

    Constructor Init(MaxKeys : Word; BitsPerKey : Byte);
    { Initialize dictionary, specify maximum keys and bits per key. }

    Constructor RestoreDictionary(FileName : String);
    { Restore dictionary saved on disk by SaveDictionary }

    { Note: Use either Init or RestoreDictionary, not both. }

    Destructor Done;
    { Release storage allocated to dictionary. }

    Function DictionarySize : Word;
    { Returns number of bytes that will be written by SaveDictionary. }

    Procedure SaveDictionary(FileName : String);
    { Save dictionary in a disk file. }

    Function InsertString(Var s : String) : Boolean;
    { Insert string in dictionary; returns TRUE if string is already there. }

    Function StringInDictionary(Var s : String) : Boolean;
    { Returns TRUE if string is in dictionary. }

    Function InsertBlock(Var Data; Len : Word) : Boolean;
    { Insert block in dictionary; returns TRUE if block is already there. }

    Function BlockInDictionary(Var Data; Len : Word) : Boolean;
    { Returns TRUE if block is in dictionary. }

    Function InsertHash(Hash : LongInt) : Boolean;
    { Insert hash in dictionary; returns TRUE if hash is already there. }

    Function HashInDictionary(Hash : LongInt) : Boolean;
    { Returns TRUE if hash is in dictionary. }

    Function EstError : Real;
    { Returns estimated probability of error. }

    Function ActError : Real;
    { Returns actual probability of error (slow, counts bits). }

  End;

Function DictionaryBytes(MaxKeys : LongInt; BitsPerKey : Byte) : LongInt;
{ Returns the size in bytes of the optimal dictionary bit table for the
  indicated key and bit-per-key counts. }

Function DictHash(Var Data; Len : Word) : LongInt;
{ Hash data block to a positive long integer. }

Implementation

Const
  MagicNumber = $E501205F; { Used to validate dictionary save file }
  RandMult = 16807;        { =7**5; RandMult must be expressable in 16 bits.
                             48271 may give better "randomness" (see ACM ref.) }
  ShuffleBits = 3;
  ShuffleShift = 16 - ShuffleBits;
  ShufTableEnd = $FFFF Shr ShuffleShift;
  HashSeed : Word = 26;    { Initial hash seed }
  RandSeed : LongInt = 1;  { Random number seed: 0 < RandSeed < 2**31-1 }

Type
  SaveFileHeader = Record
  { Header for dictionary save file (all numbers are byte-reversed) }
    Magic : LongInt;       { Magic number for validity test }
    BitsCount : LongInt;   { Bits-per-key and entry count }
    Size : Word;           { Size of dictionary bit map in bytes }
  End;

Var
  ShufTable : Array[0..ShufTableEnd] Of LongInt;
  NextOut : Word;

Function IRand : LongInt;
{ Return next "minimal standard", 31 bit pseudo-random integer.  This function
  actually computes (RandSeed * RandMult) Mod (2**31-1) where RandMult is
  a 16 bit quantity and RandSeed is 32 bits (See Carta, CACM 1/90). }
Inline(
  $A1/>RandSeed+2/       {       mov     ax,[>RandSeed+2]}
  $BF/>RandMult/         {       mov     di,>RandMult}
  $F7/$E7/               {       mul     di}
  $89/$C3/               {       mov     bx,ax}
  $89/$D1/               {       mov     cx,dx}
  $A1/>RandSeed/         {       mov     ax,[>RandSeed]}
  $F7/$E7/               {       mul     di}
  $01/$DA/               {       add     dx,bx}
  $83/$D1/$00/           {       adc     cx,0 ; cx:dx:ax = Seed * Mult }
  $D0/$E6/               {       shl     dh,1 ; split p & q at 31 bits }
  $D1/$D1/               {       rcl     cx,1}
  $D0/$EE/               {       shr     dh,1 ; cx = p, dx:ax = q }
  $01/$C8/               {       add     ax,cx}
  $83/$D2/$00/           {       adc     dx,0 ; dx:ax = p + q }
  $71/$09/               {       jno     done}
  $05/$01/$00/           {       add     ax,1 ; overflow, inc(p + q) }
  $83/$D2/$00/           {       adc     dx,0}
  $80/$E6/$7F/           {       and     dh,$7F ; limit to 31 bits }
                         {done:}
  $A3/>RandSeed/         {       mov     [>RandSeed],ax}
  $89/$16/>RandSeed+2);  {       mov     [>RandSeed+2],dx}

Function Hash(Seed : LongInt; Var Data; Len : Word) : LongInt;
{ Hash a block of data into a random long integer.  This is actually
  equivalent to the following:

     RandSeed := Seed;
     Hash := 0;
     For i := 1 To Len Do Hash := Hash + (IRand * (Data[i] + $FF00);
     Hash := Hash AND $7FFFFFFF;
     If Hash = 0 Then Inc(Hash);

  Overflow is ignored.  The seed is kept in registers; RandSeed is not
  affected by this routine. }
Inline(
  $59/                   {       pop     cx      ; cx := len}
  $5E/                   {       pop     si      ; bx:si := @data}
  $5B/                   {       pop     bx}
  $58/                   {       pop     ax      ; dx:ax := seed}
  $5A/                   {       pop     dx}
  $E3/$59/               {       jcxz    alldone}
  $FC/                   {       cld}
  $1E/                   {       push    ds}
  $8E/$DB/               {       mov     ds,bx}
  $55/                   {       push    bp}
  $31/$DB/               {       xor     bx,bx}
  $53/                   {       push    bx      ; zero accumulator}
  $53/                   {       push    bx}
  $89/$E5/               {       mov     bp,sp}
                         {next:                  ; for each byte of data...}
  $51/                   {       push    cx}
  $BF/>RandMult/         {       mov     di,>RandMult}
  $89/$C3/               {       mov     bx,ax}
  $89/$D0/               {       mov     ax,dx   ;   compute next seed}
  $F7/$E7/               {       mul     di}
  $93/                   {       xchg    ax,bx}
  $89/$D1/               {       mov     cx,dx}
  $F7/$E7/               {       mul     di}
  $01/$DA/               {       add     dx,bx}
  $83/$D1/$00/           {       adc     cx,0    ; cx:dx:ax = Seed * Mult}
  $D0/$E6/               {       shl     dh,1    ; split p & q at 31 bits}
  $D1/$D1/               {       rcl     cx,1}
  $D0/$EE/               {       shr     dh,1    ; cx = p, dx:ax = q}
  $01/$C8/               {       add     ax,cx}
  $83/$D2/$00/           {       adc     dx,0    ; dx:ax = p + q}
  $71/$09/               {       jno     noovfl}
  $05/$01/$00/           {       add     ax,1    ; overflow, inc(p + q)}
  $83/$D2/$00/           {       adc     dx,0}
  $80/$E6/$7F/           {       and     dh,$7F ; limit to 31 bits}
                         {noovfl:}
  $89/$C3/               {       mov     bx,ax   ; save seed}
  $89/$D1/               {       mov     cx,dx}
  $AC/                   {       lodsb           ; get next byte + $FF00}
  $B4/$FF/               {       mov     ah,$FF}
  $89/$C7/               {       mov     di,ax}
  $F7/$E1/               {       mul     cx      ; multiply by seed}
  $97/                   {       xchg    ax,di}
  $F7/$E3/               {       mul     bx}
  $01/$FA/               {       add     dx,di}
  $01/$46/$00/           {       add     [bp+0],ax ; accumulate}
  $11/$56/$02/           {       adc     [bp+2],dx}
  $89/$D8/               {       mov     ax,bx}
  $89/$CA/               {       mov     dx,cx}
  $59/                   {       pop     cx}
  $E2/$B9/               {       loop    next    ;  until out of data}
                         {;}
  $58/                   {       pop     ax}
  $5A/                   {       pop     dx}
  $5D/                   {       pop     bp}
  $1F/                   {       pop     ds}
  $80/$E6/$7F/           {       and     dh,$7F}
                         {alldone:}
  $89/$C3/               {       mov     bx,ax}
  $09/$D3/               {       or      bx,dx}
  $75/$01/               {       jnz     exit}
  $40);                  {       inc     ax}
                         {exit:}

Procedure Shuffle;
{ Load the shuffle table }
Begin
  For NextOut := 0 To ShufTableEnd Do ShufTable[NextOut] := IRand;
  NextOut := Word(IRand) Shr ShuffleShift;
End;

Function SIRand : LongInt;
{ Return the next shuffled random number }
Var
  y : LongInt;
Begin
  y := ShufTable[NextOut];
  ShufTable[NextOut] := IRand;
  NextOut := Word(y) Shr ShuffleShift;
  SIRand := y;
End;

Function TestBit(Var BitArray; Size : Word; BitNo : LongInt) : Boolean;
{ Returns TRUE if indicated bit number, modulo size of bit array, is set.
  Size is in bytes. }
Inline(
                         {; dx:ax := BitNo}
  $58/                   {       pop     ax}
  $5A/                   {       pop     dx}
                         {; bl := bit mask}
  $88/$C1/               {       mov     cl,al}
  $80/$E1/$07/           {       and     cl,$07}
  $B3/$80/               {       mov     bl,$80}
  $D2/$EB/               {       shr     bl,cl}
```

```
                        {; dx:ax := byte offset}
  $D1/$EA/              {       shr     dx,1}
  $D1/$D8/              {       rcr     ax,1}
  $D1/$EA/              {       shr     dx,1}
  $D1/$D8/              {       rcr     ax,1}
  $D1/$EA/              {       shr     dx,1}
  $D1/$D8/              {       rcr     ax,1}
                        {; dx := byte offset}
  $5F/                  {       pop     di}
  $39/$D7/              {       cmp     di,dx}
  $77/$0E/              {       ja      quickdiv}
                        {; protect against overflow}
  $89/$F9/              {       mov     cx,di}
                        {protloop:}
  $D1/$E1/              {       shl     cx,1}
  $39/$D1/              {       cmp     cx,dx}
  $76/$FA/              {       jbe     protloop}
  $F7/$F1/              {       div     cx}
  $89/$D0/              {       mov     ax,dx}
  $31/$D2/              {       xor     dx,dx}
                        {quickdiv:}
  $F7/$F7/              {       div     di}
                        {; es:di := seg:ofs of byte}
  $5F/                  {       pop     di}
  $01/$D7/              {       add     di,dx}
  $07/                  {       pop     es}
                        {; test bit}
  $30/$C0/              {       xor     al,al}
  $26/$22/$1D/          {       es:and  bl,[di]}
  $74/$02/              {       jz      notset}
  $FE/$C0);             {       inc     al}
                        {notset:}

Function SetBit(Var BitArray; Size : Word; BitNo : LongInt) : Boolean;
{ Sets the indicated bit number modulo size of bit array.  Returns TRUE if
  bit was already set.  Size is in bytes. }
Inline(
                        {; dx:ax := BitNo}
  $58/                  {       pop     ax}
  $5A/                  {       pop     dx}
                        {; bl := bit mask}
  $88/$C1/              {       mov     cl,al}
  $80/$E1/$07/          {       and     cl,$07}
  $B3/$80/              {       mov     bl,$80}
  $D2/$EB/              {       shr     bl,cl}
                        {; dx:ax := byte offset}
  $D1/$EA/              {       shr     dx,1}
  $D1/$D8/              {       rcr     ax,1}
  $D1/$EA/              {       shr     dx,1}
  $D1/$D8/              {       rcr     ax,1}
  $D1/$EA/              {       shr     dx,1}
  $D1/$D8/              {       rcr     ax,1}
                        {; dx := byte offset mod size }
  $5F/                  {       pop     di}
  $39/$D7/              {       cmp     di,dx}
  $77/$0E/              {       ja      quickdiv}
                        {; protect against overflow}
  $89/$F9/              {       mov     cx,di}
                        {protloop:}
  $D1/$E1/              {       shl     cx,1}
  $39/$D1/              {       cmp     cx,dx}
  $76/$FA/              {       jbe     protloop}
  $F7/$F1/              {       div     cx}
  $89/$D0/              {       mov     ax,dx}
  $31/$D2/              {       xor     dx,dx}
                        {quickdiv:}
  $F7/$F7/              {       div     di}
                        {; es:di := seg:ofs of byte}
  $5F/                  {       pop     di}
  $01/$D7/              {       add     di,dx}
  $07/                  {       pop     es}
                        {; test bit}
  $30/$C0/              {       xor     al,al}
  $88/$DC/              {       mov     ah,bl}
  $26/$22/$25/          {       es:and  ah,[di]}
  $74/$04/              {       jz      notset}
  $FE/$C0/              {       inc     al}
  $EB/$03/              {       jmp     short set}
                        {notset:}
  $26/$08/$1D);         {       es:or   [di],bl}
                        {set:}

Function LongSwap(n : LongInt) : LongInt;
{ Reverse bytes in a LongInt. }
Inline(
  $5A/                  {       pop     dx}
  $58/                  {       pop     ax}
  $86/$C4/              {       xchg    ah,al}
  $86/$D6);             {       xchg    dh,dl}

Function DictionaryBytes(MaxKeys : LongInt; BitsPerKey : Byte) : LongInt;
Begin
  DictionaryBytes := Round(MaxKeys * BitsPerKey / (-Ln(0.5) * 8));
End;

Function DictHash(Var Data; Len : Word) : LongInt;
Begin
  DictHash := Hash(Hash(HashSeed, Data, Len), Data, Len);
End;

Constructor Dictionary.Init(MaxKeys : Word; BitsPerKey : Byte);
Var
  DictBytes : LongInt;
Begin
  DictBytes := DictionaryBytes(MaxKeys, BitsPerKey);
```

(continued on page 112)

Listing One *(Listing continued, text begins on page 20.)*

```
  If DictBytes > $FFF0 Then Begin
    WriteLn(DictBytes, ' bytes optimal for dictionary, but ', $FFF0,
      ' is maximum size dictionary.  Using max size.');
    DictBytes := $FFF0;
  End Else If DictBytes > MaxAvail Then Begin
    WriteLn(DictBytes, ' bytes optimal for dictionary, but only ', MaxAvail,
      ' bytes are available.  Using ', MaxAvail);
    DictBytes := MaxAvail;
  End Else If DictBytes < 16 Then DictBytes := 16;
  DictSize := DictBytes;
  GetMem(DictArray, DictSize);
  FillChar(DictArray^, DictSize, 0);
  DictCount := 0;
  DictBits := BitsPerKey;
End;

Constructor Dictionary.RestoreDictionary(FileName : String);
Var
  Header : SaveFileHeader;
  DictBytes : LongInt;
  f : File;
  OldMode : Byte;
Begin
  OldMode := FileMode;
  FileMode := $40;
  Assign(f, FileName);
  Reset(f, 1);
  BlockRead(f, Header, SizeOf(Header));
  With Header Do Begin
    Magic := LongSwap(Magic);
    Size := Swap(Size);
    DictBytes := FileSize(f) - SizeOf(Header);
    If (Magic <> MagicNumber) Or (Size <> DictBytes) Or (Size < 16)
    Or (Size > $FFF0) Then Begin
      WriteLn('File ', FileName, ' is not a dictionary save file.');
      Halt(1);
    End;
    DictSize := Size;
    DictBits := BitsCount And $FF;
    DictCount := LongSwap(BitsCount And $FFFFFF00);
    GetMem(DictArray, DictSize);
    BlockRead(f, DictArray^, DictSize);
    Close(f);
    FileMode := OldMode;
  End;
End;

Destructor Dictionary.Done;
Begin
  FreeMem(DictArray, DictSize);
  DictArray := Nil;
  DictSize := 0;
  DictBits := 0;
  DictCount := 0;
End;

Function Dictionary.DictionarySize : Word;
Begin
  DictionarySize := DictSize + SizeOf(SaveFileHeader);
End;

Function Dictionary.InsertString(Var s : String) : Boolean;
Begin
  InsertString := InsertBlock(s[1], Length(s));
End;

Function Dictionary.StringInDictionary(Var s : String) : Boolean;
Begin
  StringInDictionary := BlockInDictionary(s[1], Length(s));
End;

Function Dictionary.InsertBlock(Var Data; Len : Word) : Boolean;
Begin
  InsertBlock := InsertHash(DictHash(Data, Len));
End;

Function Dictionary.BlockInDictionary(Var Data; Len : Word) : Boolean;
Begin
  BlockInDictionary := HashInDictionary(DictHash(Data, Len));
End;

Function Dictionary.InsertHash(Hash : LongInt) : Boolean;
Var
  i : Byte;
  InDict : Boolean;
Begin
  InDict := True;
  RandSeed := Hash;
  Shuffle;
  For i := 1 To DictBits Do
    If Not SetBit(DictArray^, DictSize, SIRand) Then InDict := False;
  If Not InDict Then Inc(DictCount);
  InsertHash := InDict;
End;

Function Dictionary.HashInDictionary(Hash : LongInt) : Boolean;
Var
  i : Byte;
  InDict : Boolean;
Begin
  InDict := True;
  RandSeed := Hash;
  Shuffle;
  i := 0;
  While (i < DictBits) And InDict Do Begin
    If Not TestBit(DictArray^, DictSize, SIRand) Then InDict := False;
    Inc(i);
  End;
  HashInDictionary := InDict;
End;

Procedure Dictionary.SaveDictionary(FileName : String);
Var
  Header : SaveFileHeader;
  f : File;
Begin
  Assign(f, FileName);
  ReWrite(f, 1);
  With Header Do Begin
    Magic := LongSwap(MagicNumber);
    Size := Swap(DictSize);
    BitsCount := LongSwap(DictCount) + DictBits;
  End;
  BlockWrite(f, Header, SizeOf(Header));
  BlockWrite(f, DictArray^, DictSize);
  Close(f);
End;

Function Dictionary.EstError : Real;
Begin
  EstError := Exp(Ln(1.0-Exp(-(DictCount*DictBits)/(DictSize*8.0)))*DictBits);
End;

Function Dictionary.ActError : Real;
Var
  AllBits, BitsOn, i : LongInt;
Begin
  AllBits := LongInt(DictSize) * 8;
  BitsOn := 0;
  For i := 0 To Pred(AllBits) Do
    If TestBit(DictArray^, DictSize, i) Then Inc(BitsOn);
  ActError := Exp(Ln(BitsOn / AllBits) * DictBits);
End;

End.
```

End Listing

Listing One *(Text begins on page 36.)*

```
#ifndef STRING_H
#define STRING_H

#include <string.h>
#include <memory.h>
#include <malloc.h>

#ifdef NULL
# undef NULL
#endif
#define NULL 0

typedef enum { False, True } Boolean;

// a String is a simple implementation of a C++ string class
class String
{
    char     *s;        // actual pointer to text
  public:
    String(void)           { s = strdup(""); }
    String(char *c)        { s = strdup(c); }
    String(char *c, int n) { s = new char[n+1]; memcpy(s,c,n); s[n]=0; }
    String(String &ss, char *c) { s = malloc(strlen(ss.s)+strlen(c)+1);
                                  strcpy(s,ss.s); strcat(s,c); }
    String(String &ss)     { s = strdup(ss.s); }
    ~String(void)          { delete s; }
    operator char *(void)  { return s; }
    String& operator +(char *c)  { String *a = new String(*this,c); return *a;}
    String& operator +=(char *c) { *this = *this + c; return *this; }
    String& operator =(char *c)  { delete s; s = strdup(c); return *this; }
    String& operator =(String& a){ delete s; s = strdup(a.s); return *this; }
    operator ==(char *c) const;
    operator ==(String *c) const;
#ifdef DEBUG
    void dump() const;
    Boolean verify() const;
#endif /*DEBUG*/
};

// a StringListElement is a single item in a StringList
class StringListElement
{
    String s;                   // this String in the list
    StringListElement *next;    // pointer to next element in list
  public:
    StringListElement(void)      : next(NULL), s("")  {}
    StringListElement(char *c)   : next(NULL), s(c)   {}
    StringListElement(char *c, int n) : next(NULL), s(c,n) {}
    StringListElement(String &ss)     : next(NULL), s(ss)  {}
    ~StringListElement(void)     { if (next) delete next; next=NULL; }
    friend class StringList;
    friend class StringListIterator;
#ifdef DEBUG
    void dump() const;
    Boolean verify() const;
#endif /*DEBUG*/
};

// a StringList is a simple single-linked list of strings
class StringList
{
    StringListElement *head;    // first String in list
    StringListElement *tail;    // last String in list
  public:
    StringList(void) : head(NULL), tail(NULL) {}
    StringList(String& ss) { head = tail = new StringListElement(ss); }
    StringList(char *ss)   { head = tail = new StringListElement(ss); }
    ~StringList(void)      { if (head) { delete head; head=NULL; } }
    String& find(char *s) const;
    void clear(void)       { delete head; head = tail = NULL; }
    StringList& operator +=(StringList& xx);
    StringList& operator +=(char *ss);
    StringList& operator +=(String& ss);
    StringList& operator =(StringList& xx);
    operator int();
    friend class StringListIterator;
#ifdef DEBUG
    void dump() const;
    Boolean verify() const;
#endif /*DEBUG*/
};

// a StringListIterator is a method of traversing a list of strings
class StringListIterator
{
    const StringList *list;     // StringList to be traversed
    StringListElement *nxt;     // current String in StringList
  public:
    StringListIterator(void) : list(NULL), nxt(NULL) {}
    StringListIterator(const StringList *l) : list(l), nxt(l->head) {}
    String *next(void);
    void reset()         { nxt = (list != NULL) ? list->head: NULL; }
    int anymore() const { return (nxt != NULL); }
    StringListIterator& operator =(const StringList& ss);
#ifdef DEBUG
    void dump() const;
    Boolean verify() const;
#endif /*DEBUG*/
};
#endif // STRING_H
```

End Listing One

Listing Two

```
#include <stream.h>
#include "String.h"

int main (int, char *[])
{
    String a("Hello ");
    String *b = new String("world.");
    String c;

    c = a + *b + "\n";

    cout << "a + b = " << (char *)c;
    cout << "a = " << (char *)a << "\n";
    cout << "b = " << (char *)*b << "\n";

    StringList l(a);
    l += *b;

    l.dump();
}
```

End Listing Two

Listing Three

```
#ifndef ASSERT_HDR
#define ASSERT_HDR

#ifdef DEBUG
extern void _assertRtn(char *, int);

# define assert(condition) \
    if (condition) ; else _assertRtn(__FILE__,__LINE__);

#else /*ifndef DEBUG*/

# define assert(condition)

#endif

#endif /*ASSERT_HDR*/
```

End Listing Three

Listing Four

```
#include <stream.h>

#ifdef DEBUG
void _assertRtn(char *file, int line)
{
    cerr << "\nAssertion Failure in file '" << file
                                  << "' line " << line << "\n";
    line = 0; line /= line;    // force core dump
}
#endif
```

End Listing Four

Listing Five

```
#include "String.h"
#include "Assert.h"

#ifdef DEBUG
# include <stream.h>
#endif

/*****  String class  ******/
// String comparison operator
String::operator ==(char *c) const
{
    assert(this->verify());

    // compare String to char array
    return strcmp(s,c) == 0;
}

// String comparison operator
String::operator ==(String *c) const
{
    assert(this->verify());
    // compare String to String
    return strcmp(s,(char *)c) == 0;
}
#ifdef DEBUG
void String::dump(void) const
{
    assert(this->verify());
    cerr << "String(\"" << s << "\")";
}
Boolean String::verify(void) const
{
    // Strings must always point to something.
    if (s == NULL) return False;
    return True;
}
```

```
#endif

/****** StringList class (and StringListElement)  ******/
StringList& StringList::operator +=(String& ss)
{
    assert(this->verify());
    if (tail)
    {
        tail->next = new StringListElement(ss);
        tail = tail->next;
    }
    else
        head = tail = new StringListElement(ss);
    return *this;
}
StringList& StringList::operator +=(char *ss)
{
    assert(this->verify());
    if (tail)
    {
        tail->next = new StringListElement(ss);
        tail = tail->next;
    }
    else
        head = tail = new StringListElement(ss);
    return *this;
}
StringList& StringList::operator +=(StringList& xx)
{
    assert(this->verify());
    // add new list to old list item by item
    for (StringListElement *le=xx.head; le; le=le->next)
        *this += le->s;
    return *this;
}
// StringList assignment operator (performs deep copy)
StringList& StringList::operator =(StringList& xx)
{
    assert(this->verify());
    // get rid of old list
    clear();
    // add new list to (clear) old list item by item
    for (StringListElement *le=xx.head; le; le=le->next)
        *this += le->s;
    // return new copy of old list
    return *this;
}
// (int)(StringList) cast returns number of strings in list
StringList::operator int()
{
    int count = 0;
    StringListIterator ll(this);

    assert(this->verify());
    while (ll.next() != NULL)
        count++;
    return count;
}
#ifdef DEBUG
//
//        dump() - display instance in format "StringList(...)"
//
void StringList::dump(void) const
{
    // check consistancy
    assert(this->verify());
    // print header
    cerr << "StringList(";
    // use StringListElement::dump() to recursively display all members
    if (head != NULL) head->dump();
    // print trailer
    cerr << ")\n";
}
Boolean StringList::verify(void) const
{
    // if there are elements in this list, ensure they are valid.
    // (note that head->verify() ensures the entire list is valid.)
    if ((head!=NULL) && !head->verify()) return False;

    // Both the head and tail must either be null or non-null.
    if ((head!=NULL) && (tail==NULL)) return False;
    if ((head==NULL) && (tail!=NULL)) return False;

    return True;
}
#endif /*DEBUG*/

#ifdef DEBUG
void StringListElement::dump(void) const
{
    assert(this->verify());
    s.dump();
    if (next != NULL)
    {
        cerr << ',';
        next->dump();
    }
}
Boolean StringListElement::verify(void) const
{
    // An element of a list of Strings must point to a valid String.
    if ((char *)(s) == NULL) return False;
    if (!s.verify()) return False;
    // If there is another element within this list, it must be valid.
    if ((next!=NULL) && !next->verify()) return False;

    return True;
}
#endif

/****** StringListIterator class  ******/

// assignment operator
StringListIterator& StringListIterator::operator =(const StringList& ss)
{
    assert(this->verify());
    list = &ss;
    nxt = ss.head;
    return *this;
}
// get next item in list of strings pointed to by iterator
String *StringListIterator::next(void)
{
    assert(this->verify());
    if (list == NULL) return NULL;    // no StringList, so no next item
    if (nxt == NULL) return NULL;     // at end of list, so no next item
    String *aa = &(nxt->s);           // save pointer to String
    nxt = nxt->next;                  // point to next item in list
    return aa;                        // return pointer to String
}
#ifdef DEBUG
Boolean StringListIterator::verify(void) const
{
    // if there is a list available, verify it.
    if (!list->verify()) return False;

    // if we haven't reached the end of the list,
    // verify the next element
    if (nxt != NULL && !nxt->verify()) return False;

    // everything appears correct
    return True;
}
#endif /*DEBUG*/
```

End Listing Five

Listing Six

```
CC = CC
CFLAGS = -DDEBUG

prog : main.o String.o lib.o
    $(CC) main.o String.o lib.o -o prog

String.o : String.C String.h
    $(CC) -c $(CFLAGS) String.C

main.o : main.C String.h
    $(CC) -c $(CFLAGS) main.C

lib.o : lib.C
    $(CC) -c $(CFLAGS) lib.C

clean :
    rm -f *.o a.out core

clobber : clean
    rm -f prog
```

End Listings

Listing One *(Text begins on page 44.)*

```
/** CTrace.h -- Definitions for using the Trace class **/
#define _H_CTrace

/* System/library header files */
#include <Commands.h>           /* standard menu command definition */
#include <oops.h>               /* standard OOP definitions */
#include <stdarg.h>             /* varg macro definitions */
#include <CDesktop.h>           /* definitions for desktop class */
#include <CBartender.h>         /* definitions for menu bar manager */
#include <CDataFile.h>          /* definitions for data file class */
#include <CApplication.h>       /* definitions for the application class */
#include <CDocument.h>          /* definitions for parent class */
#include <Constants.h>          /* miscellaneous environment constants */

/* Local header files */
#include "CLogPanorama.h"       /* definitions for logging panorama class */

/* Resource numbers */
#define TRACE_MENU_ID     (2000)      /* menu resource ID */
#define TRACE_MENU_SHOW   (2000L)     /* menu command for show/hide log */
#define TRACE_MENU_MASK   (2001L)     /* menu command for log masking */
#define TRACE_WINDOW_ID   (2000)      /* main window resource ID */
#define TRACE_MASK_DIALOG (2000)      /* resource ID for dialog box */
#define FIRST_MASK         (3)        /* item # of first checkbox in dialog */
#define LAST_MASK          (34)       /* item # of last checkbox in dialog */
#define UNHILITE_CONTROL   (255)      /* magic part # for disabling control */
#define OKAY_BUTTON_ITEM    (1)       /* item # for the 'okay' button */
#define CANCEL_BUTTON_ITEM  (2)       /* item # for the 'cancel' button */

/* Standard trace categories */
#define T_ERROR          (0x00000001)          /* serious error */
#define T_WARNING        (0x00000002)          /* mildly serious problem */
#define T_INFO           (0x00000004)          /* news you can use */
#define T_FUNC_IN        (0x00000008)          /* function entry */
#define T_FUNC_OUT       (0x00000010)          /* function exit */

/* Other constants */
#define MAX_USER_BUFF (MAX_LOGREC_CHAR-19+1)  /* max length of user message */
#define TRACE_DEFAULT_MASK      (0L)          /* initial trace mask */

/* External references */
extern CDesktop         *gDesktop;            /* the whole desktop view */
extern CApplication     *gApplication;        /* the application object */
extern CBartender       *gBartender;          /* the menu bar object */
extern OSType           gSignature;           /* application signature */
struct CTrace : CDocument
        {
        /* local instance variables */
        CLogPanorama  *itsLogPanorama; /* panorama for trace messages */
        unsigned long currMask;        /* currently enabled trace categories */
        /* local class methods */
        void    ITrace(short records);
        void    ToggleTraceWindow(void);
        void    SetTraceMask (void);
        void    Trace (unsigned long mask, char *format, ...);
        Boolean IsItVisible(void);
        /* inherited methods overriden */
        void    UpdateMenus (void);
        Boolean DoSaveAs (SFReply *macSFReply);
        Boolean Close (Boolean quitting);
        };
```

End Listing One

Listing Two

```
/** CTrace.c -- Methods for the trace document class. **/
#include "CTrace.h"             /* trace class parameters */

/** Global declaration **/
CTrace
        *gTrace;                /* the one instance of this class */

/** ITrace() -- Initializes trace document object. **/
void CTrace::ITrace
        (
        short records           /* number of records before wrap */
        )
{
Rect
        frameRect;              /* window frame */
CDocument::IDocument (gApplication, TRUE);
itsWindow = new (CWindow);
itsWindow->IWindow (TRACE_WINDOW_ID, FALSE, gDesktop, this);
itsWindow->GetFrame (&frameRect);
itsWindow->Move (gDesktop->bounds.right - frameRect.right - RIGHT_SMARGIN,
                 gDesktop->bounds.top + TOP_SMARGIN);
itsLogPanorama = new (CLogPanorama);
itsLogPanorama->ILogPane (records, this, itsWindow);
itsMainPane = itsLogPanorama;
currMask = TRACE_DEFAULT_MASK;
gTrace = this;
}

/** ToggleTraceWindow() -- Toggles visibility of trace window. **/
void CTrace::ToggleTraceWindow(void)
{
if (itsWindow->visible)
        itsWindow->Hide ();
else
        {
        itsWindow->Show ();
        itsWindow->Select ();
        }
}

/** Close() -- Overrides normal document method by closing trace window. **/
Boolean CTrace::Close
        (
        Boolean quitting                        /* ignored */
        )
{
itsWindow->Hide ();
return (TRUE);
}

/** SetTraceMask -- Allows user to set/clear defined trace masks.**/
void CTrace::SetTraceMask (void)
{
int
        bitNum,                 /* bit number within the trace mask */
        checkBoxState,          /* state of a checkbox (0=unset,1=set) */
        item,                   /* loop counter */
        itemType,               /* item type (4=button, 5=checkbox) */
        whichItem;              /* item number selected by user */
Handle
        itemStuff;              /* handle to dialog item parameters */
Boolean
        done;                   /* loop-termination flag */
Str255
        title;                  /* text associated with a dialog item */
Rect
        itemRect;               /* rectangle surrounding a control */
DialogPtr
        maskDialog;             /* structure for dialog box */
/* Pull up the mask dialog box out of the resource fork */
maskDialog = GetNewDialog (TRACE_MASK_DIALOG, NULL, (Ptr)(-1));

/* Run through checkboxes */
for (item=FIRST_MASK, bitNum=0;
                item<=LAST_MASK; item++,
                bitNum++)
        {
        GetDItem (maskDialog, item, &itemType, &itemStuff, &itemRect);
        GetCTitle ( (ControlHandle)itemStuff, title);
        PtoCstr ((char*)title);
        if (strcmp((char*)title, "Undefined") != 0)
                {
                checkBoxState = ((currMask&(1L<<bitNum)) == 0L) ? 0 : 1;
                SetCtlValue ( (ControlHandle) itemStuff, checkBoxState);
                }
        else
                HiliteControl (itemStuff, UNHILITE_CONTROL);
        }
/* The default button (#1) is the okay button, draw outline around it. */
GetDItem (maskDialog, OKAY_BUTTON_ITEM, &itemType, &itemStuff, &itemRect);
SetPort (maskDialog);
PenSize (3, 3);
InsetRect (&itemRect, -4, -4);
FrameRoundRect (&itemRect, 16, 16);

/* Get events from dialog manager and process accordingly */
done = FALSE;
while (!done)
        {
        ModalDialog (NULL, &whichItem);
        GetDItem (maskDialog, whichItem, &itemType, &itemStuff, &itemRect);
        switch (itemType)
                {
                case ctrlItem + btnCtrl :       /* CANCEL or OKAY */
                        if (whichItem == OKAY_BUTTON_ITEM)
                        {
                        currMask = 0L;
                    for (item=FIRST_MASK, bitNum=0; item<=LAST_MASK; item++, bitNum++)
                        {
                        GetDItem (maskDialog, item, &itemType, &itemStuff, &itemRect);
                             checkBoxState = GetCtlValue ( (ControlHandle) itemStuff);
                                  currMask |= (checkBoxState==0) ? 0L : (1L<<bitNum);
                        }
                        done = TRUE;
                        }
                        else if (whichItem == CANCEL_BUTTON_ITEM)
                        done = TRUE;
                        break;
                case ctrlItem + chkCtrl :       /* a category checkbox */
                        checkBoxState = GetCtlValue ( (ControlHandle) itemStuff);
                                if (checkBoxState == 0)
                        SetCtlValue ( (ControlHandle) itemStuff, 1);
                                else
                        SetCtlValue ( (ControlHandle) itemStuff, 0);
                                break;
                        default:
                                break;
                        }
        }

/* On exit, trash dialog record and controls */
DisposDialog (maskDialog);
}
/** Trace() -- Checks current mask **/
void CTrace::Trace
        (
        unsigned long mask,             /* severity of message */
        char *format,                   /* format for user's arguments */
        ...                             /* arguments to format (varg) */
        )
{
static char
        traceBuff[MAX_LOGREC_CHAR],     /* string that will be added to log */
        userBuff[MAX_LOGREC_CHAR*2],    /* user's contribution to log record */
        prefix[40];                     /* date+time string */
int
        maxUserBuff;            /* maximum length of formatted user string */
long
        timeSecs;               /* current time/date */
DateTimeRec
        dateRec;                /* time/date in MM/DD/YY HH:MM:SS components */
/* Should the message get added to the trace log? */
if ( (currMask & mask) != 0)
        {
        /* format the user's portion of the message */
        vsprintf (userBuff, format, __va(format));
```

(continued on page 118)

Listing Two *(Listing continued, text begins on page 44.)*

```
        /* make sure the entire record will fit into traceBuff */
        if (strlen (userBuff) > MAX_USER_BUFF)
                userBuff[MAX_USER_BUFF] = NULL;
        /* build log message and add it to the log */
        GetDateTime (&timeSecs);
        Secs2Date (timeSecs, &dateRec);
        sprintf (traceBuff, "%02d/%02d/%02d--%02d:%02d:%02d %s",
                        dateRec.month, dateRec.day, dateRec.year-1900,
                        dateRec.hour, dateRec.minute, dateRec.second,
                        userBuff);
        itsLogPanorama->AddString (traceBuff);
        }
}

/** IsItVisible() -- Returns 'visible' flag to update menu bar entries. **/
Boolean CTrace::IsItVisible(void)
{
return (itsWindow->visible);
}

/** UpdateMenus() -- Disables Save and Revert entries **/
void CTrace::UpdateMenus(void)
{
inherited::UpdateMenus ();;
gBartender->DisableCmd (cmdSave);
gBartender->DisableCmd (cmdRevert);
}

/** DoSaveAs() -- Writes out contents of itsLogList to indicated file. **/
Boolean CTrace::DoSaveAs
        (
        SFReply *macSFReply             /* the user's choice of file */
        )
{
char
        logRecBuff[MAX_LOGREC_CHAR];    /* buffer for log entry */
short
        maxRec,                         /* number of records in LogList */
        offsetToNull,                   /* byte offset to end of log entry */
        rec;                            /* loop counter */
/* Dispose of the data used for the old file record */
if (itsFile != NULL)
        itsFile->Dispose ();
/* Set up the new data file (no error checking!) */
itsFile = new (CDataFile);
((CDataFile *)itsFile)->IDataFile ();
itsFile->SFSpecify (macSFReply);
itsFile->CreateNew (gSignature, 'TEXT');
itsFile->Open (fsRdWrPerm);

/*  Write out all records in list (add carriage return to end of each line).*/
maxRec = (short)(itsLogPanorama->itsLogList)->GetNumItems();
for (rec=1; rec<=maxRec; rec++)
        {
        (itsLogPanorama->itsLogList)->GetString (rec, logRecBuff);
        offsetToNull = strlen (logRecBuff);
        logRecBuff[offsetToNull] = '\r';
        ((CDataFile*)itsFile)->WriteSome (logRecBuff, offsetToNull+1);
        }
return (TRUE);
}
```

End Listing Two

Listing Three

```
/** CLogPanarama.h -- Definitions for using the LogPanorama class **/
#define _H_CLogPanorama

/* System/library headers */
#include <CPanorama.h>          /* definitions for superclass Panorama */
#include <CScrollPane.h>        /* definitions for ScrollPane class */
#include <CWindow.h>            /* definitions for Window class */
#include <oops.h>               /* standard OOP definitions */
#include <Constants.h>          /* miscellaneous look-n-feel paramaters */
#include <Limits.h>             /* numeric extrema */

/* Local headers */
#include "CLogList.h"           /* definitions for LogList class */
#define LOGPANE_FONT    (monaco)    /* font family of text in the log window */
#define LOGPANE_FONT_SIZE       (9) /* size of text in the log window */
#define LOGPANE_HORZ_SCROLL     (5) /* units per horizontal scroll */
#define LOGPANE_VERT_SCROLL     (1) /* units per vertical scroll */
#define LOGPANE_INSET           (4) /* left margin for start of text */

/* Externals referenced */
extern RgnHandle
        gUtilRgn;                       /* drawing region */
struct CLogPanorama : CPanorama
        {
        /* local class instance data */
        CLogList        *itsLogList;    /* the buffer for logged data */
        /* local class methods */
        void    ILogPane (short records, CBureaucrat *aSupervisor,
                                                CWindow *anEnclosure);
        void    AddString (char *theString);
        /* inherited methods overriden */
        void    Draw (Rect *theRect);
        };
```

End Listing Three

Listing Four

```
/** CLogPanorama.c -- Methods for a CLogPanorama class object. **/
#include "CLogPanorama.h"                               /* defines log class */

/** ILogPanorama -- Initializes an instance of the log panorama class. **/
void CLogPanorama::ILogPane
        (
        short records,                  /* number of records in LogList */
        CBureaucrat     *aSupervisor,   /* in-charge for this panorama */
        CWindow *aWindow                /* window object to place pane into */
        )
{
FontInfo
        fontParms;                      /* paramaters of selected font */
short
        lineSpace,                      /* pixels per line */
        charSpace;                      /* pixels per widest character */
Rect
        maxWindowRect,                  /* maximum growth of log window */
        marginRect;                     /* inside margins of viewable area */
CScrollPane
        *theScrollPane;                 /* pane associated with panorama */

/* Set drawing parameters and adjust record size, if necessary. **/
aWindow->Prepare ();
TextFont (LOGPANE_FONT);
TextSize (LOGPANE_FONT_SIZE);
GetFontInfo (&fontParms);
lineSpace = fontParms.ascent+fontParms.descent+fontParms.leading;
charSpace = fontParms.widMax;
if ( ((long)records*(long)lineSpace) > (long)INT_MAX)
        records = INT_MAX / lineSpace;
SetRect (&maxWindowRect, MIN_WSIZE, MIN_WSIZE,
                (MAX_LOGREC_CHAR * charSpace) + SBARSIZE,
                (records * lineSpace) + SBARSIZE);
aWindow->SetSizeRect (&maxWindowRect);

/* Initialize Panorama's ScrollPane, set scroll units to the defaulted
** values, and attach the Panorama to the ScrollPane. */
theScrollPane = new (CScrollPane);
theScrollPane->IScrollPane(aWindow, this, 0, 0, 0, 0,
                sizELASTIC, sizELASTIC,TRUE, TRUE, TRUE);
theScrollPane->FitToEnclFrame (TRUE, TRUE);
theScrollPane->SetSteps (LOGPANE_HORZ_SCROLL, LOGPANE_VERT_SCROLL);

/* Initialize Panarama to include maximum chars wide and maximum records tall,
** set the Panarama units to one char wide and one char tall. */
CPanorama::IPanorama(theScrollPane, aSupervisor, MAX_LOGREC_CHAR,
                records, 0, 0, sizELASTIC, sizELASTIC);
SetScales (charSpace, lineSpace);
FitToEnclosure (TRUE, TRUE);
theScrollPane->InstallPanorama (this);

/*  Create the LogList and initialize. */
itsLogList = new (CLogList);
itsLogList->ILogList (records);
}

/** Draw() -- Refreshes the visible portion of the window. **/
void CLogPanorama::Draw
        (
        Rect *drawRect          /* portion of window to refresh */
        )
{
short
        firstRec,               /* record number of first visible line */
        hScale,                 /* how many pixels wide is a character? */
        lastRec,                /* record number of last line */
        totalRec,               /* total number of records in LogList */
        vScale;                 /* how many pixels tall is a line? */
register short
        currRow,                /* window coordinates of current row */
        rec;                    /* loop counter */
char
        buff[MAX_LOGREC_CHAR];  /* buffer for fetching log records */
/*  First, translate draw rectangle to records. **/
GetScales (&hScale, &vScale);
totalRec = (short)itsLogList->GetNumItems ();
firstRec = (drawRect->top / vScale);
if (firstRec == 0)
        firstRec = 1;
lastRec = (drawRect->bottom / vScale) + 1;
if (lastRec > totalRec)
        lastRec = totalRec;
/*  Refresh all of visible lines. **/
Prepare ();
for (rec=firstRec, currRow=firstRec*vScale;
                rec<=lastRec;
                rec++, currRow+=vScale)
        {
        itsLogList->GetString (rec, buff);
        MoveTo (LOGPANE_INSET, currRow);
        DrawString (CtoPstr(buff));
        }
}

/** AddString() -- Adds a new string to panorama. **/
void CLogPanorama::AddString
        (
        char *theString         /* null-terminated string to add */
        )
{
Rect
        frameRect;              /* interior of current frame */
short
        listLimits,             /* maximum the LogList will hold */
        hSpan,                  /* the horizontal span of frame */
        vSpan,                  /* the vertical span of frame */
        hScale,                 /* horizontal pixels in panarama unit */
        vScale;                 /* vertical pixels in panarama unit */
Point
        topRecord,              /* LogList record number of top row */
        bottomRecord,           /* LogList record number of bottom row */
        newRecord,              /* LogList record number of new row */
        currPosition,           /* panorama coordinates of top/left frame */
        recPosition;            /* panorama coordinates of new string */
/*  Add the record to the LogList. */
```

(continued on page 120)

Listing Four *(Listing continued, text begins on page 44.)*

```
itsLogList->AddString (theString);
/*  Get coordinates of current frame and calculate tentative coordinates for
newly added record. */
GetPosition (&currPosition);
GetFrameSpan (&hSpan, &vSpan);
GetScales (&hScale, &vScale);
topRecord.v = currPosition.v + 1;
bottomRecord.v = topRecord.v + vSpan - 1;
newRecord.v = (short)itsLogList->GetNumItems ();
newRecord.h = currPosition.h;

/* Determine where we are in reference to bottom of screen and of list. **/
if (newRecord.v > (bottomRecord.v+1) )
        {
        /* bottom record isn't visible */
        currPosition.v = newRecord.v - vSpan;
        ScrollTo (currPosition, FALSE);
        GetInterior (&frameRect);
        Prepare ();
        EraseRect (&frameRect);
        }
else
        {
        /* bottom record is visible */
        listLimits = itsLogList->GetMaxRecordCount ();
        if (bottomRecord.v < listLimits)
                {
                /* room in list--create blank line if necessary */
                if (newRecord.v == (bottomRecord.v + 1) )
                        {
                        Scroll (0, 1, TRUE);
                SetRect (&frameRect, newRecord.h*hScale, (newRecord.v-1)*vScale,
                                (newRecord.h+hSpan)*hScale, (newRecord.v)*vScale);
                        }
                else
                SetRect (&frameRect, newRecord.h*hScale, (newRecord.v-1)*vScale,
                                (newRecord.h+hSpan-1)*hScale, (newRecord.v)*vScale);
                }
        else if (bottomRecord.v > listLimits)
                {
                currPosition.v = newRecord.v - vSpan;
                ScrollTo (currPosition, FALSE);
                GetInterior (&frameRect);
                Prepare ();
                EraseRect (&frameRect);
                }
        else
                {
          /* bottom of pane=limit of list, so do our own scrolling */
                Prepare ();
                GetInterior (&frameRect);
                ScrollRect (&frameRect, 0, -vScale, gUtilRgn);
         SetRect (&frameRect, bottomRecord.h*hScale, (bottomRecord.v-1)*vScale,
                        (bottomRecord.h+hSpan-1)*hScale, (bottomRecord.v)*vScale);
                EraseRect (&frameRect);
                }
        }
Draw (&frameRect);
itsScrollPane->Calibrate();
}
```

End Listing Four

Listing Five

```
/** CLogList.h -- Definitions for a LogList object. **/
#define _H_CLogList

#include <CList.h>              /* definitions for superclass */
#include <oops.h>               /* standard OOP definitions */
#include <string.h>             /* miscellaneous string definitions */

#define MAX_LOGREC_CHAR (200L)  /* size of longest entry (inc NULL) */

struct CLogList : CList
        {
        /* internal instance data */
        short           maxRec;         /* maximum number of records */
        /* local class methods */
        void    ILogList (short records);
        void    AddString (char *theString);
        void    GetString (short which, char *theString);
        short   GetMaxRecordCount (void);
        /* inherited methods overriden */
        void    Dispose (void);
        };
```

End Listing Five

Listing Six

```
/** CLogList.c -- Methods for a LogList object. **/

#include "CLogList.h"           /* definitions for LogList class */

/** ILogList -- Initializes a LogList for the indicated number of entries. **/
void CLogList::ILogList
        (
        short records           /* maximum number of entries */
        )
{
CList::IList ();
maxRec = records;
}

/** Dispose -- Frees all records (and their handles) in the list **/
void CLogList::Dispose (void)
{
short
        i;                              /* loop counter */
Handle
        record;                         /* handle to list record */
while (GetNumItems() > 0)
        {
        record = (Handle)FirstItem();
        Remove ((CObject*)record);
        DisposHandle (record);
        }
}

/** AddString -- Adds a string to LogList. **/
void CLogList::AddString
        (
        char *theString         /* pointer to null-terminated string */
        )
{
Handle
        record;                 /* handle for a list entry */
if (strlen(theString)+1 < MAX_LOGREC_CHAR)
        {
        record = NewHandle (strlen(theString)+1);
        strcpy (*record, theString);
        }
else
        {
        record = NewHandle (MAX_LOGREC_CHAR);
        strncpy (*record, theString, MAX_LOGREC_CHAR);
        *record[MAX_LOGREC_CHAR-1] = NULL;
        }
Append ((CObject*)record);
if (GetNumItems () > maxRec)
        {
        record = (Handle)FirstItem ();
        Remove ((CObject*)record);
        DisposHandle (record);
        }
}

/** GetString -- Grabs requested entry and copies it to user's buffer. **/
void CLogList::GetString
        (
        short which,            /* record number to return */
        char *theString         /* point to destination buffer */
        )
{
Handle
        record;                 /* handle for a list entry */
if ( (record=(Handle)NthItem(which)) != NULL)
        strcpy (theString, *record);
else
        theString[0] = 0;
}

/** GetMaxRecordCount -- Returns max.number of records available in LogList **/
short CLogList::GetMaxRecordCount (void)
{
return (maxRec);
  }
```

End Listings

Listing Four *(Text begins on page 74.)*

```
;*********************************************************************
;* PROT - A 386 protected mode DOS extender                          *
;* Copyright (C) 1989, by Al Williams  All rights reserved.          *
;* Permission is granted for non-commercial use of this software     *
;* subject to certain conditions (see PROT.ASM).                     *
;* This file is: STACKS.INC, which contains all the stack segments.  *
;*********************************************************************
; 16 bit stack segment (for CODE16)
SSEG        SEGMENT  PARA STACK 'STACK' USE16
SSEG0       DB       DOSSTACK DUP (?)
SSEG1       EQU      $
SSEG        ENDS

; 16 bit stack segment for vm86 int (both hardware & INT 30)
SSINT       SEGMENT  PARA STACK 'STACK' USE16
SSINT0      DB       VM86STACK DUP (?)
SSINT1      EQU      $
SSINT       ENDS

; private stack for default critical error handler dos calls
CSTACK      SEGMENT  PARA STACK 'STACK' USE16
            DB       CRITSTACK DUP (?)
CSTACK      ENDS

; 32 bit stack segment
SS32        SEGMENT  PARA PUBLIC 'STACK' USE32
SSEG32      DB       PMSTACK DUP (?)
SSEG321     EQU      $
SS32        ENDS
```

End Listing Four

Listing Five

```
;*********************************************************************
;* PROT - A 386 protected mode DOS extender                          *
;* Copyright (C) 1989, by Al Williams  All rights reserved.          *
;* Permission is granted for non-commercial use of this software     *
;* subject to certain conditions (see PROT.ASM).                     *
;* This file is: INT386.INC                                          *
;*********************************************************************

; Peculiarities: 1. We don't emulate lock, IRETD, PUSHFD, POPFD yet; 2. When
;   calling INT 25 or INT 26 from protected mode flags are destroyed (not left
;   on stack as in VM86, real mode); 3. For now I don't support adding offsets
;   to the return address on your vm86 stack to change where IRET goes to.
;   That could be fixed, but I don't know of any PC software that does that

; fake segment for far ret interrupts (this has no descriptor in GDT/LDT)
QISR        SEGMENT  PARA PUBLIC 'CODE16' USE16
            ASSUME   CS:QISR
; push sacrifical words for IRET to eat -- PL0 stack controls return anyway
QIRET:
            PUSH     0
            PUSH     0
            PUSH     0
            IRET
QISR        ENDS

; IDT segment
IDTABLE     SEGMENT  PARA PUBLIC 'DATA32' USE32
IDTBEG      EQU      $
            DQ       TOPINT+1 DUP (0)
IDTEND      EQU      $
IDTABLE     ENDS

;ISR segment
DEFINT      MACRO    N
INT&N       LABEL    FAR
            PUSH     &N
            JMP      NEAR PTR INTDUMP
            ENDM

ISR         SEGMENT  PARA PUBLIC 'CODE32' USE32
            ASSUME   CS:ISR
ISRBEG      EQU      $
; Defines interrupt handlers from 0 to TOPINT (TOPINT defined in EQUMAC.INC)
INTNO       =        0
            REPT     TOPINT+1
            DEFINT   %INTNO
INTNO       =        INTNO + 1
            ENDM
; Debug dump messages
MESSAREA    DB       'INT=',0
STKM        DB       'Stack Dump:',0
TASKM       DB       ' TR=',0
RTABLE      DB       'G'
            DB       'F'
            DB       'D'
            DB       'E'
GTABLE      DB       'DISISBPSPBXDXCXAX'
MEMMESS     DB       'Memory Dump:',0
; All interrupts come here. We check for the interrupt # pushed on the stack
; and vector. Adds interrupt latency, but simplifies IDT construction.
INTDUMP     LABEL    NEAR
; check for GP error
            CMP      BYTE PTR [ESP],0DH
            JZ       NEAR PTR INT13H

NOT13:
; check for vm86 psuedo-int
            CMP      BYTE PTR [ESP],30H
            JZ       NEAR PTR INT30H
; hardware interrupt?
            CMP      BYTE PTR [ESP],20H
            JB       SHORT NOTIO
IF          ATCLASS
            CMP      BYTE PTR [ESP],2FH
ELSE
            CMP      BYTE PTR [ESP],27H
ENDIF
            JA       SHORT NOTIO
            JMP      NEAR PTR HWINT
NOTIO:
; An unexpected interrupt; crank out a debug dump, exit to dos
            PUSHAD
            PUSH     GS
            PUSH     FS
            PUSH     DS
            PUSH     ES
            MOV      AX,SEL_VIDEO
            MOV      ES,AX
            MOV      AX,CS
            MOV      DS,AX
; do dump
            MOV      ECX,4
INTL1:
            MOV      AL,[RTABLE-1+ECX]
            CALL32F  SEL_CODE32,OUCH
            MOV      AL,'S'
            CALL32F  SEL_CODE32,OUCH
            MOV      AL,'='
            CALL32F  SEL_CODE32,OUCH
            POP      EAX
            CALL32F  SEL_CODE32,HEXOUT2
            PUSH     ECX
            MOV      ECX,6
LSP1:       MOV      AL,' '
            CALL32F  SEL_CODE32,OUCH
            LOOP     LSP1
            POP      ECX
            LOOP     INTL1
            CALL32F  SEL_CODE32,CRLF
            XOR      ECX,ECX
INTL2:      CMP      CL,5
            JNZ      SHORT NOCRINT
            CALL32F  SEL_CODE32,CRLF
NOCRINT:
            MOV      AL,'E'
            CALL32F  SEL_CODE32,OUCH
            MOV      AL,[GTABLE+ECX*2]
            CALL32F  SEL_CODE32,OUCH
            MOV      AL,[GTABLE+1+ECX*2]
            CALL32F  SEL_CODE32,OUCH
            MOV      AL,'='
            CALL32F  SEL_CODE32,OUCH
            POP      EAX
            CALL32F  SEL_CODE32,HEXOUT4
            MOV      AL,' '
            CALL32F  SEL_CODE32,OUCH
            INC      CL
            CMP      CL,8
            JNE      SHORT INTL2
            MOV      EBX,OFFSET MESSAREA
            CALL32F  SEL_CODE32,MESSOUT
            POP      EAX
            CALL32F  SEL_CODE32,HEXOUT
            MOV      EBX,OFFSET TASKM
            CALL32F  SEL_CODE32,MESSOUT
            STR      AX
            CALL32F  SEL_CODE32,HEXOUT2
            CALL32F  SEL_CODE32,CRLF
; stack dump
            XOR      EAX,EAX
            MOV      AX,SS
            LSL      EDX,EAX
            JNZ      SHORT INTABT
            MOV      EBX,OFFSET STKM
            CALL32F  SEL_CODE32,MESSOUT
            XOR      CL,CL
INTL3:      CMP      ESP,EDX
            JAE      SHORT INTABT
            TEST     CL,7
            JNZ      SHORT NOSCR
            CALL32F  SEL_CODE32,CRLF
NOSCR:      POP      EAX
            CALL32F  SEL_CODE32,HEXOUT4
            INC      CL
            MOV      AL,' '
            CALL32F  SEL_CODE32,OUCH
            JMP      SHORT INTL3

INTABT:
; Check for memory dump request
            MOV      AX,SEL_DATA
            MOV      DS,AX
            ASSUME   DS:DAT32
            MOV      AX,WORD PTR DUMP_SEG
            OR       AX,AX
            JZ       SHORT NOMEMDUMP
; come here to do memory dump
            CALL32F  SEL_CODE32,CRLF
            PUSH     DS
            PUSH     CS
            POP      DS
            MOV      EBX,OFFSET MEMMESS
            CALL32F  SEL_CODE32,MESSOUT
            CALL32F  SEL_CODE32,CRLF
            POP      DS
            MOV      AX,WORD PTR DUMP_SEG
            MOV      ES,AX
            CALL32F  SEL_CODE32,HEXOUT2
            MOV      AL,':'
            CALL32F  SEL_CODE32,OUCH
            MOV      EDX,DUMP_OFF
```

```
                MOV     EAX,EDX
                CALL32F SEL_CODE32,HEXOUT4
                MOV     ECX,DUMP_CNT
DUMPLOOP:
                MOV     AL,' '
                CALL32F SEL_CODE32,OUCH
                MOV     EAX,ES:[EDX] ; get word
                CALL32F SEL_CODE32,HEXOUT4
                ADD     EDX,4
                SUB     ECX,4
                JA      SHORT DUMPLOOP
                CALL32F SEL_CODE32,CRLF
NOMEMDUMP:
                MOV     AL,20H          ; Send EOI signal
IF              ATCLASS
                OUT     0A0H,AL
ENDIF
                OUT     20H,AL          ; just in case hardware did it
                MOV     AL,7FH          ; return 7f to DOS
                BACK2DOS

; Check the GP fault. If mode isn't VM86, do debug dump. Otherwise
; emulate an instruction. If instruction isn't known, do debug dump.
INT13H:
                ADD     ESP,4           ; balance stack (remove intno)
                TEST    [ESP+12],20000H
                JZ      SHORT SIM13A ; wasn't a vm86 interrupt!
                ADD     ESP,4           ; remove error code
                PUSH    EAX
                PUSH    EBX
                PUSH    DS
                PUSH    EBP
                MOV     EBP,ESP         ; point to stack frame
                ADD     EBP,10H
                MOV     AX,SEL_DATA0
                MOV     DS,AX
                MOV     EBX,[EBP+4]  ; get cs
                AND     EBX,0FFFFH
                SHL     EBX,4
                ADD     EBX,[EBP]       ; get eip
                XOR     EAX,EAX         ; al = OPCODE byte
                                        ; ah = # of bytes skipped over
                                        ; bit 31 of eax=1 if OPSIZ prefix
                                        ; encountered
                JMP     SHORT INLOOP
; set sign bit of eax if OPSIZ
FSET:           OR      EAX,80000000H
INLOOP:         MOV     AL,[EBX]
                INC     AH
                INC     EBX
                CMP     AL,66H          ; opsize prefix
                JZ      SHORT FSET
; scan for instructions
                CMP     AL,9DH
                JZ      SHORT DOPOPF
                CMP     AL,9CH
                JZ      SHORT DOPUSHF
                CMP     AL,0FAH
                JZ      NEAR PTR DOCLI
                CMP     AL,0FBH
                JZ      NEAR PTR DOSTI
                CMP     AL,0CDH
                JZ      NEAR PTR DOINTNN
                CMP     AL,0CFH
                JZ      NEAR PTR DOIRET
                CMP     AL,0F0H
                JZ      NEAR PTR DOLOCK
; Whoops! What the $#$%$#! is that?
                POP     EBP
                POP     DS
                POP     EBX
                POP     EAX
SIM13:
                PUSH    0               ; simulate error
SIM13A:
                PUSH    13              ; simulate errno
                JMP32S  NOT13

;*****************************************************************
; The following routines emulate VM86 instructions. Entry conditions:
; eax[31]=1 iff opsiz preceeded instruction
; ah=count to adjust eip on stack
; al=instruction
; [EBX] next opcode byte
; ds: zerobase segment

; This routine emulates a popf
DOPOPF:
                MOV     BX, [EBP]    ; fix IP
                ADD     BL,AH
                ADC     BH,0
                MOV     [EBP],BX
; get ss*10H, add esp fetch top of stack
                MOVZX   EBX,WORD PTR [EBP+10H]
                SHL     EBX,4
                ADD     EBX,[EBP+0CH]
                MOVZX   EAX,WORD PTR [EBX]
                MOV     EBX,[EBP+8]  ; get his real flags
                AND     BX,07000H    ; only preserve NT,IOPL
                AND     AX,08FFFH    ; wipe NT,IOPL in new flags
                OR      EAX,EBX
                MOV     [EBP+8],EAX  ; save his real flag image
                MOV     EBX,2
                ADD     [EBP+0CH],EBX
                MOV     EBX,0FFFEFFFFH
                AND     [EBP+8],EBX
                POP     EBP
                POP     DS
                POP     EBX
```

(continued on page 124)

Listing Five *(Listing continued, text begins on page 74.)*

```
            POP     EAX
            IRETD
; Routine to emulate pushf
DOPUSHF:
            MOV     BX, [EBP]    ; Fix ip
            ADD     BL,AH
            ADC     BH,0
            MOV     [EBP],BX
            MOV     EAX,[EBP+8]  ; get his flags
; get ss, add esp and "push" flags
            MOVZX   EBX,WORD PTR [EBP+10H]
            SHL     EBX,4
            ADD     EBX,[EBP+0CH]
            MOV     [EBX-2],AX
            MOV     EBX,2
; adjust stack
            SUB     [EBP+0CH],EBX
; mask out flag bits
            MOV     EBX,0FFFEFFFFH
            AND     [EBP+8],EBX
            POP     EBP
            POP     DS
            POP     EBX
            POP     EAX
            IRETD
; Emulate CLI
DOCLI:
            MOV     BX, [EBP]    ; Fix ip
            ADD     BL,AH
            ADC     BH,0
            MOV     [EBP],BX
            MOV     EAX,[EBP+8]  ; get flags
            OR      EAX,20000H   ; set vm, clr RF & IOPL
            AND     EAX,0FFFECDFFH
            MOV     [EBP+8],EAX  ; replace flags
            POP     EBP
            POP     DS
            POP     EBX
            POP     EAX
            IRETD
; Emulate STI
DOSTI:
            MOV     BX, [EBP]    ; Fix ip
            ADD     BL,AH
            ADC     BH,0
            MOV     [EBP],BX
            MOV     EAX,[EBP+8]  ; get flags
            OR      EAX,20200H   ; set vm, clr RF & IOPL
            AND     EAX,0FFFECFFFH
            MOV     [EBP+8],EAX  ; replace flags
            POP     EBP
            POP     DS
            POP     EBX
            POP     EAX
            IRETD
; This routine emulates an INT nn instruction
DOINTNN:
            PUSH    EDX
            PUSH    ECX
; get ss
            MOVZX   EDX,WORD PTR [EBP+10H]
            SHL     EDX,4
; add esp
            ADD     EDX,[EBP+0CH]
; move flags, qsir address to vm86 stack & correct esp
; ... flags
            MOV     CX, [EBP+08H]
            MOV     [EDX-2],CX
            MOV     WORD PTR [EDX-4],SEG QIRET
            MOV     WORD PTR [EDX-6],OFFSET QIRET
            SUB     DWORD PTR [EBP+0CH],6
            MOV     CX, [EBP]    ; ip
            INC     AH           ; adjust ip by # of bytes to skip
            ADD     CL,AH
            ADC     CH,0
            MOV     [EBP],CX
; get tss alias (always directly above TSS in GDT)
            STR     DX           ; get our task #
            SUB     DX,8         ; alias is one above
            MOV     ES,DX
            MOV     DX,SEL_DATA
            MOV     DS,DX
            ASSUME  DS:DAT32
; get pl0 esp from TSS & push to local stack
            MOV     EDX,INTSP
            SUB     EDX,4
            MOV     INTSP,EDX
            MOV     ECX,ES:[4]   ; esp0
            MOV     [EDX],ECX
; get int vector
            MOV     DX,SEL_DATA0
            MOV     DS,DX
            MOV     ECX,ESP      ; adjust stack for int 30H
            ADD     ECX,60
            MOV     ES:[4],ECX
; test for zero; if so called from int 30H
            OR      AH,AH
            MOVZX   EDX,AL
            JZ      SHORT FROM30
; otherwise get int vector from CS:EIP stream
            MOVZX   EDX,BYTE PTR [EBX]
            MOV     ECX,ESP
            ADD     ECX,24
            MOV     ES:[4],ECX   ; adjust stack for non-int 30H
FROM30:
; interrupt vector*4 = VM86 interrupt vector address
            SHL     EDX,2
; try to clean up mess on stack
            MOV     AX,SEL_DATA
            MOV     DS,AX
            MOV     STO2,EDX
            POP     ECX
            POP     EDX
            XCHG    STO2,EDX
            MOV     STO1,ECX
            MOV     STO3,EBP
            POP     EBP
            XCHG    STO3,EBP
            POP     ECX
            MOV     BX,SEL_DATA
            MOV     DS,BX
            MOV     STO4,ECX
            POP     EBX
            POP     EAX
            MOV     CX,SEL_DATA0
            MOV     DS,CX
; copy segment registers & esp for vm86 int
            PUSH    [EBP+20H]
            PUSH    [EBP+1CH]
            PUSH    [EBP+18H]
            PUSH    [EBP+14H]
            PUSH    [EBP+10H]
            PUSH    [EBP+0CH]
            MOV     ECX,[EBP+08]
; push flags (with vm=1,iopl=0),cs, eip, rf=0
            OR      ECX,20000H
; clear iopl, rf, tf, if and push flags
            AND     ECX,0FFFECCFFH
            PUSH    ECX
; read new cs/ip from 8086 idt
; ... push CS
            MOVZX   ECX,WORD PTR [EDX+2]
            PUSH    ECX
; ... push IP
            MOVZX   ECX,WORD PTR [EDX]
            PUSH    ECX
            MOV     CX,SEL_DATA
            MOV     DS,CX
            PUSH    STO4
            MOV     ECX,STO1
            MOV     EDX,STO2
            MOV     EBP,STO3
            POP     DS
            IRETD                ; go on to vm86 land
; Emulate IRET instruction
DOIRET:
; vm86 stack
            MOVZX   EAX,WORD PTR[EBP+10H]
            SHL     EAX,4
            ADD     EAX,[EBP+0CH]
            MOV     EBX,[EAX]    ; get cs:ip
; If top of stack=0:0 than a RETF or RETF 2 was detected
            OR      EBX,EBX
            JZ      SHORT FARRETINT
            PUSH    ECX
            XOR     ECX,ECX
; compare return address with QIRET
            MOV     CX, SEG QIRET
            SHL     ECX,16
            MOV     CX,OFFSET QIRET
            CMP     EBX,ECX
            POP     ECX
; if equal than "normal" IRET
            JZ      SHORT NORMIRET
; Not equal then that vm86 jerk is "faking" an IRET to pass control
; We must build a "fake" pl0 frame
; adjust sp
            ADD     DWORD PTR [EBP+0CH],6
; get ip
            MOVZX   EBX,WORD PTR [EAX]
            MOV     [EBP],EBX
; get cs
            MOVZX   EBX,WORD PTR [EAX+2]
            MOV     [EBP+4],EBX
; get new flags
            MOVZX   EBX,WORD PTR [EAX+4]
            OR      EBX,20000H   ; set vm, clr RF & IOPL
            AND     EBX,0FFFECFFFH
            MOV     [EBP+8],EBX
            POP     EBP
            POP     DS
            POP     EBX
            POP     EAX
            IRETD                ; go on
; this means qiret caught a FAR RET instead of an IRET; preserve current flags!
FARRETINT:
            MOV     EAX,EBP
            POP     EBP
            POP     DS
            PUSH    EBP
            PUSH    EAX
            MOV     BX,DS
            MOV     AX,SEL_DATA
            MOV     DS,AX
            MOV     STO3,EBX
            POP     EBP          ; ISR's ebp
            MOV     EAX,[EBP+0CH]
            ADD     EAX,6        ; skip pushes from qiret
            MOV     STO4,EAX
; get flags
            MOV     EAX,[EBP+08H]
            MOV     STO2,EAX
            JMP     SHORT NIRET
; This handles the "normal" case
NORMIRET:
            MOV     BX,[EAX+4]   ; get flags
            MOV     EAX,EBP
            POP     EBP
            POP     DS
```

```
                PUSH    EBP
                PUSH    EAX
                MOV     AX,BX
                MOV     BX,DS
                PUSH    SEL_DATA
                POP     DS
                MOV     STO2,EAX
                MOV     STO3,EBX
                POP     EBP             ; ISR's ebp
                XOR     EAX,EAX
                MOV     STO4,EAX
NIRET:
                PUSH    ESI
                XOR     ESI,ESI
                OR      DWORD PTR [EBP+28H],0
; if CS=0 then int 30H asked  for segment save
                JNZ     SHORT V86IRET
                MOV     EAX,[EBP+14H]
                MOV     SAV_ES,EAX
                MOV     EAX,[EBP+18H]
                MOV     SAV_DS,EAX
                MOV     EAX,[EBP+1CH]
                MOV     SAV_FS,EAX
                MOV     EAX,[EBP+20H]
                MOV     SAV_GS,EAX
                MOV     ESI,8
V86IRET:
                MOV     AX,ES
                MOV     STO1,EAX
                POP     EBP
                XCHG    EBP,[ESP]
; get tss alias
                STR     AX
                SUB     AX,8
                MOV     ES,AX
                ASSUME  DS:DAT32
                MOV     EAX,ES:[4]    ; get our current stack begin
; see if we have to balance the VM86 stack
                TEST    SS:[EAX+ESI+8],20000H
                JZ      SHORT STKADJD
                MOV     EBX,STO4
                OR      EBX,EBX
                JZ      SHORT ADJSTK
; balance vm86 stack
                MOV     SS:[EAX+ESI+0CH], EBX
                JMP     SHORT STKADJD
ADJSTK:         ADD     DWORD PTR SS:[EAX+ESI+0CH],6
STKADJD:
; get quasi flags
                MOV     EBX,STO2
; get real flags
                PUSH    SS:[EAX+ESI+8]
; preserve flags
                MOV     DWORD PTR SS:[EAX+ESI+8],EBX
LEAVEFLAGS:
; only let 8086 part of flags stay
                AND     DWORD PTR SS:[EAX+ESI+08],01FFFH
                POP     EBX             ; load real flags into ebx
; save 386 portion of old flags (AND IP)
                AND     EBX,0FFFFE200H
                OR      SS:[EAX+ESI+8],EBX
                POP     ESI
                XCHG    EAX,[ESP]
                PUSH    EAX             ; stack = ebx, new sp
                MOV     EBX,INTSP
; get prior pl0 esp from local stack
                MOV     EAX,[EBX]
                ADD     EBX,4
                MOV     INTSP,EBX
                MOV     ES:[4],EAX    ; restore to TSS
; restore registers
                POP     EBX
                MOV     ES,WORD PTR STO1
                MOV     DS,WORD PTR STO3
                POP     EAX             ; restore "real" eax
                XCHG    EAX,[ESP]
                POP     ESP             ; set up new top stack
                XCHG    EAX,[ESP+4]
                OR      EAX,EAX       ; test cs
                XCHG    EAX,[ESP+4]
                JNZ     SHORT GOIRET
                ADD     ESP,8         ; skip fake CS/IP from INT 30H
GOIRET:
; reset resume flag
                AND     DWORD PTR [ESP+8],0FFFECFFFH
                IRETD
; Emulate lock prefix
DOLOCK:
                POP     EBP
                POP     DS
                POP     EBX
                POP     EAX
                PUSH    0FFFFH
                PUSH    13            ; simulate errno
                JMP32S  NOT13

; This is interface routine for protected mode program call VM86 interrupts.
; Call with es:ebx pointing to a parameter block
; +00 flag - if 1 then resave ES, DS, FS & GS into parameter block after call
; +04 int number (0-255)    (required)
; +08 eflags
; +12 vm86 esp     (required)
; +16 vm86 ss      (required)
; +20 vm86 es
; +24 vm86 ds
; +28 vm86 fs
; +32 vm86 gs
; +36 vm86 ebp  ( to replace that used in call )
; +40 vm86 ebx  ( to replace that used in call )
```

(continued on page 126)

Listing Five *(Listing continued, text begins on page 74.)*

```
; All other registers will be passed to vm86 routine
; This routine depends on the dointnn routine

INT30H:
            ADD     ESP,4           ; remove intno
            CMP     BYTE PTR ES:[EBX],0
            JZ      SHORT NOSEGSAV
; dummy CS/IP to signal IRET to save segments
            PUSH    0
            PUSH    0
NOSEGSAV:
            PUSH    ES:[EBX+32]  ; stack up registers
            PUSH    ES:[EBX+28]
            PUSH    ES:[EBX+24]
            PUSH    ES:[EBX+20]
            PUSH    ES:[EBX+16]
            PUSH    ES:[EBX+12]
; force VM86=1 in EFLAGS
            XCHG    EAX,ES:[EBX+8]
            OR      EAX,20000H
            AND     EAX,0FFFECFFFH
            PUSH    EAX
            XCHG    EAX,ES:[EBX+8]
            PUSH    0               ; don't care cs
            PUSH    0               ; don't care eip
            MOV     EBP,ESP
            PUSH    EAX
            PUSH    ES:[EBX+40]  ; vm86 ebx
            PUSH    DS
            PUSH    ES:[EBX+36]  ; vm86 ebp
            MOV     AX,SEL_DATA0
            MOV     DS,AX
; get user's intno
            MOV     AL,ES:[EBX+4]
; set flag to dointnn not to check cs:ip for int #
            MOV     AH,0FFH
; go ahead.... make my interrupt
            JMP32S  DOINTNN
; Handle hardware int! This routine uses INT 30 to handle HW interrupts
; If interrupted in protected mode, a special stack
; is used. If in VM86 mode, the current VM86 stack is used
HWINT:
            XCHG    EAX,[ESP]    ; swap eax & int #
            PUSH    DS
            PUSH    ES
            PUSH    EBX
            MOV     BX,SEL_DATA
            MOV     DS,BX
            MOV     ES,BX
            CMP     EAX,28H
            JB      SHORT IRQ07
            ADD     EAX,48H      ; vector IRQ8-F to INT 70-77
            JMP     SHORT IRQSET

IRQ07:
            SUB     EAX,24       ; vector IRQ0-7 to INT 8-0F
IRQSET:
; set up special interrupt frame
            MOV     HINTFRAME.VMINT,EAX
            MOV     HINTFRAME.VMEBP,EBP
            POP     EBX
            MOV     HINTFRAME.VMEBX,EBX
            PUSH    EBX
            MOV     EAX,020000H  ; model flags
            MOV     HINTFRAME.VMFLAGS,EAX
            MOV     EAX,OFFSET SSINT1
            MOV     HINTFRAME.VMESP,EAX
            MOV     AX,SEG SSINT1
            MOV     HINTFRAME.VMSS,EAX
            MOV     EAX,[ESP+24] ; get flags
            TEST    EAX,20000H   ; check vm
            JZ      SHORT NOTVMHW
            MOV     EAX,[ESP+28] ; get vm86's esp
            MOV     HINTFRAME.VMESP,EAX
            MOV     EAX,[ESP+32]
            MOV     HINTFRAME.VMSS,EAX
NOTVMHW:
            MOV     EBX,OFFSET HINTFRAME
            PUSH    FS
            PUSH    GS
            INT     30H          ; Do interrupt
            POP     GS
            POP     FS
            POP     EBX
            POP     ES
            POP     DS
            POP     EAX
            IRETD
ISREND      EQU     $
ISR         ENDS
```

End Listing Five

Listing Six

```
;*****************************************************************
;* PROT - A 386 protected mode DOS extender                      *
;* Copyright (C) 1989, by Al Williams. All rights reserved.      *
;* Permission is granted for non-commercial use of this software *
;* subject to certain conditions (see PROT.ASM).                 *
;* This file is: TSS.INC, the Task State Segment definitions.    *
;*****************************************************************
; Define TSS structure. For more details refer to the Intel documentation.
; Remember, defined values are only defaults, and can be changed when
; a value is defined
TSSBLK      STRUC
BLINK       DD      0
ESPP0       DD      OFFSET SSEG321
SSP0        DD      SEL_STACK
ESPP1       DD      0
SSP1        DD      SEL_STACK
ESPP2       DD      0
SSP2        DD      SEL_STACK
CR31        DD      0
EIP1        DD      OFFSET USER
EF1         DD      200H
EAX1        DD      0
ECX1        DD      0
EDX1        DD      0
EBX1        DD      0
ESP1        DD      OFFSET SSEG321
EBP1        DD      0
ESI1        DD      0
EDI1        DD      0
ES1         DD      SEL_DATA
CS1         DD      SEL_UCODE
SS1         DD      SEL_STACK
DS1         DD      SEL_UDATA
FS1         DD      SEL_DATA0
GS1         DD      SEL_VIDEO
LDT1        DD      0
            DW      0
IOT         DW      $+2-OFFSET BLINK
IOP         DB      8192 DUP (0)
            DB      0FFH
TSSBLK      ENDS

TSSSEG      SEGMENT  PARA PUBLIC 'DATA32' USE16
            ORG     0
; Dummy TSS that stores the original machine state
TSS0BEG     TSSBLK  <>
TSS0END     EQU     $

; TSS to run the USER task
TSS1BEG     TSSBLK  <>
TSS1END     EQU     $

TSSSEG      ENDS
```

End Listing Six

Listing Seven

```
;*********************************************************************
;* PROT - A 386 protected mode DOS extender                          *
;* Copyright (C) 1989, by Al Williams                                *
;* All rights reserved.                                              *
;* Permission is granted for non-commercial use of this software     *
;* subject to certain conditions (see PROT.ASM).                     *
;* This file is: CODE16.INC, the 16 bit DOS entry/exit code.         *
;*********************************************************************

CSEG            SEGMENT  PARA PUBLIC 'CODE16' USE16
                ASSUME   CS:CSEG, DS:CSEG
BEG16           EQU      $
IDTSAV          DF       0              ; space to save old real mode IDT
XZRO            DF       0              ; Zero constant for inhibiting IDT

TEMP            EQU      THIS FWORD     ; Space to load GDT
TLIM            DW       (GDTEND-GDT)-1
TEMD            DD       0

; area to save stack pointer
SOFFSAV         DW       0
SSEGSAV         DW       0

; old keyboard interrupt vector -- we have to catch reboot requests
KEYCHAIN        EQU      THIS DWORD
KEYOFF          DW       ?
KEYSEG          DW       ?

INTM            DB       0              ; interrupt mask - pic 1
IF              ATCLASS
INTMAT          DB       0              ; interrupt mask - pic 2 (AT ONLY)
ENDIF

;psp
PSP             DW       0

; error messages
NOT386M         DB       'Error: this program requires an 80386 or 80486'
                DB       ' processor.',13,10,'$'
VM86M           DB       'Error: this program will not execute '
                DB       'in VM86 mode.'
                DB       13,10,'$'

; 16 bit ss/sp for return to real mode
LOAD16          DD       OFFSET SSEG1-1
                DW       SEL_RDATA

;****** Begin program
ENTRY           LABEL    FAR
START           PROC     NEAR
                PUSH     CS             ; set up DS segment, save PSP
                POP      DS
                MOV      AX,ES
                MOV      PSP,AX         ; save PSP
                MOV      BX,DAT32
                MOV      ES,BX
                MOV      ES:_PSP,AX
; check to see if we are running on a 386/486
                XOR      AX,AX
                PUSH     AX
                POPF
                PUSHF
                POP      AX
                AND      AX,0F000H
                CMP      AX,0F000H
                JNZ      SHORT NOT86
NOT386:
                MOV      DX, OFFSET NOT386M
NOT386EXIT:
                MOV      AH,9
                INT      21H
                MOV      AX,4C80H
                INT      21H            ; exit
NOT86:
                MOV      AX,0F000H
                PUSH     AX
                POPF
                PUSHF
                POP      AX
                AND      AX,0F000H
                JZ       SHORT NOT386
; If we got here we are on an 80386. Check PM flag
                SMSW     AX
                AND      AX,1           ; are we in protected mode?
                MOV      DX,OFFSET VM86M
                JNZ      SHORT NOT386EXIT
; OK.. clear to proceed. Set up new ^C, keyboard and Critical error handlers
                MOV      AX,3509H
                INT      21H
                MOV      AX,ES
                MOV      KEYSEG,AX
                MOV      KEYOFF,BX
                MOV      AX,2509H
                MOV      DX,OFFSET REBOOT
                INT      21H
                MOV      AX,2523H
                MOV      DX,OFFSET CTRLC
                INT      21H
                MOV      AX,2524H
                MOV      DX,OFFSET CRITERR
                INT      21H
; * Create segments
                PUSH     GDTSEG
                POP      ES
```

(continued on page 128)

Listing Seven *(Listing continued, text begins on page 74.)*

```
        MOV     EDX, OFFSET GDT
        MOV     EBX,CS
        SHL     EBX,4           ; calc segment base address
        MOV     ECX,0FFFFH      ; 64 K limit (don't change)
        MOV     AH,ER_CODE      ; read/exec code seg
        XOR     AL,AL           ; size
        PUSH    GDTSEG
        POP     ES
        MOV     EDX, OFFSET GDT
        MOV     SI,SEL_CODE16
        CALL    MAKE_DESC       ; make code seg (16 bit/real)
        MOV     ECX,0FFFFFH
        XOR     EBX,EBX
        MOV     SI,SEL_DATA0
        XOR     ECX,ECX
        DEC     ECX             ; ecx=ffffffff
        MOV     AL,1
        MOV     AH,RW_DATA
        CALL    MAKE_DESC       ; make data  ( 4G @ zero base )
        XOR     EAX,EAX
        INT     12H
        MOVZX   ECX,AX
        SHL     ECX,10
        LOADFREE BX             ; get free memory segment
        SUB     ECX,EBX
        DEC     ECX
        MOV     SI,SEL_FREE
        MOV     AL,1
        MOV     AH,RW_DATA
        CALL    MAKE_DESC
        XOR     EAX,EAX
        MOV     AH,88H          ; get top of extended memory
        INT     15H
        SHL     EAX,10          ; * 1024
        OR      EAX,EAX         ; any extended present?
        MOV     ECX,EAX
        JNZ     SHORT EXTPRES
        MOV     ECX,1
EXTPRES:
        DEC     ECX
        MOV     EBX,100000H
        MOV     SI,SEL_EXT      ; 0 limit segment if no ext.
        MOV     AL,1
        MOV     AH,RW_DATA
        CALL    MAKE_DESC
        XOR     EBX,EBX
        MOV     BX,SEG SEG32ENT
        SHL     EBX,4
        MOV     ECX,(SEG32END-SEG32BEG)-1
        MOV     AH,ER_CODE
        MOV     AL,1
        MOV     SI,SEL_CODE32
        CALL    MAKE_DESC       ; 32 bit code segment
        XOR     EBX,EBX
        MOV     BX,USERCODE
        SHL     EBX,4
        MOV     ECX,(USERCODEEND-USERCODEBEG)-1
        MOV     AH,ER_CODE
        MOV     AL,1
        MOV     SI,SEL_UCODE
        CALL    MAKE_DESC
        XOR     EBX,EBX
        MOV     BX,USERDATA
        SHL     EBX,4
        MOV     ECX,(USERDATAEND-USERDATABEG)-1
        MOV     AH,RW_DATA
        MOV     AL,1
        MOV     SI,SEL_UDATA
        CALL    MAKE_DESC
        XOR     EBX,EBX
        MOV     BX,SS32
        SHL     EBX,4           ; always para align stacks!
        MOV     ECX,(SSEG321-SSEG32)-1
        MOV     AH,RW_DATA      ; stack seg is data type
        MOV     AL,1
        MOV     SI,SEL_STACK
        CALL    MAKE_DESC
        XOR     EBX,EBX
        MOV     BX, SSEG
        SHL     EBX,4
        MOV     ECX,0FFFFH      ; real mode limit (don't change)
        XOR     AL,AL
        MOV     AH,RW_DATA
        MOV     SI,SEL_RDATA
        CALL    MAKE_DESC       ; 16 bit data for return to r/m
        XOR     EBX,EBX
        MOV     BX,SEG GDT
        SHL     EBX,4
        ADD     EBX,OFFSET GDT
        MOV     ECX,(GDTEND-GDT)-1
        MOV     AL,1
        MOV     AH,RW_DATA
        MOV     SI,SEL_GDT
        CALL    MAKE_DESC
        MOV     AX,500H         ; set video to page 0
        INT     10H
        MOV     AH,0FH
        INT     10H             ; get mode
        MOV     EBX,0B0000H     ; monochrome
        CMP     AL,7            ; check for mono
        JZ      SHORT VIDEOCONT
        MOV     EBX,0B8000H
VIDEOCONT:
        MOV     ECX,3999        ; limit for text page
        MOV     AL,1
        MOV     AH,RW_DATA
        MOV     SI,SEL_VIDEO
        CALL    MAKE_DESC       ; make video segment
        XOR     EBX,EBX
        MOV     BX,DAT32
        SHL     EBX,4
        MOV     ECX,(DAT32END-DAT32BEG)-1
        MOV     AH,RW_DATA
        MOV     AL,1
        MOV     SI,SEL_DATA
        CALL    MAKE_DESC
        XOR     EBX,EBX
        MOV     BX,IDTABLE
        SHL     EBX,4
        MOV     ECX,(IDTEND-IDTBEG)-1
        MOV     AH,RW_DATA
        MOV     AL,1
        MOV     SI,SEL_IDT
        CALL    MAKE_DESC
        XOR     EBX,EBX
        MOV     BX,ISR
        SHL     EBX,4
        MOV     ECX,(ISREND-ISRBEG)-1
        MOV     AH,ER_CODE
        MOV     AL,1
        MOV     SI,SEL_ICODE
        CALL    MAKE_DESC
        XOR     EBX,EBX
        MOV     BX,TSSSEG
        SHL     EBX,4
; compute TSS length
        MOV     ECX,(TSS0END-TSS0BEG)-1
        MOV     AH,RW_DATA
        MOV     AL,1
        MOV     SI,SEL_TSS0
        CALL    MAKE_DESC
        MOV     AH,TSS_DESC
        MOV     SI,TSS0
        CALL    MAKE_DESC
        ADD     EBX,OFFSET TSS1BEG
        MOV     SI,TSS1
; compute TSS length
        MOV     ECX,(TSS1END-TSS1BEG)-1
        CALL    MAKE_DESC
        MOV     SI,SEL_TSS1
        MOV     AH,RW_DATA
        CALL    MAKE_DESC
        MOVZX   EBX,PSP
        SHL     EBX,4
        MOV     ECX,255
        MOV     AH,RW_DATA
        MOV     AL,1
        MOV     SI,SEL_PSP
        CALL    MAKE_DESC
        PUSH    ES
        MOV     AX,PSP
        MOV     ES,AX
        XOR     EBX,EBX
        MOV     BX,ES:[2CH]
        MOV     AX,BX
        SHL     EBX,4
        DEC     AX
        MOV     ES,AX
        XOR     ECX,ECX
        MOV     CX,ES:[3]
        SHL     ECX,4
        DEC     ECX             ; get limit
        MOV     SI,SEL_ENV
        POP     ES
        MOV     AL,1
        MOV     AH,RW_DATA
        CALL    MAKE_DESC       ; make envriroment segment
; turn on A20
        MOV     AL,1
        CALL    SETA20
        CLI                     ; no interrupts until prot mode
        MOV     SSEGSAV,SS
; save sp for triumphant return to r/m
        MOV     SOFFSAV,SP
        SIDT    IDTSAV
        LIDT    XZRO            ; save and load IDT
        XOR     EBX,EBX
        MOV     BX,SEG GDT
        SHL     EBX,4
        ADD     EBX,OFFSET GDT
        MOV     TEMD,EBX
        LGDT    TEMP            ; set up GDT
        MOV     EAX,CR0
        OR      EAX,1           ; switch to prot mode!
        MOV     CR0,EAX
; jump to load CS and flush prefetch
        JMPABS  SEL_CODE16,PROT1
PROT1:
        OPSIZ
        JMPABS32 SEL_CODE32,SEG32ENT


; Jump here to return to real mode DOS.
; If desired AL can be set a to DOS exit code
BACK16  LABEL   FAR
        MOV     BL,AL           ; save exit code
        CLI
        XOR     EAX,EAX
        MOV     DR7,EAX         ; turn off debug (just in case)
; restore stack
        LSS     ESP,FWORD PTR CS:LOAD16
        MOV     AX,SEL_RDATA
        MOV     DS,AX
        MOV     ES,AX
        MOV     FS,AX
        MOV     GS,AX
        MOV     EAX,CR0
```

(continued on page 130)

Listing Seven *(Listing continued, text begins on page 74.)*

```
; return to real mode
            AND     EAX,07FFFFFF2H
            MOV     CR0,EAX
; jump to load CS and clear prefetch
            JMPABS  CSEG,NEXTREAL
NEXTREAL    LABEL   FAR
            MOV     AX,CS
            MOV     DS,AX
            LIDT    IDTSAV      ; restore old IDT 0(3ff)
            IN      AL,21H
            MOV     INTM,AL
; reprogram PIC's
IF          ATCLASS
            IN      AL,0A1H
            MOV     INTMAT,AL
            MOV     AL,11H
            OUT     0A0H,AL
            OUT     20H,AL
            IDELAY
            MOV     AL,70H
            OUT     0A1H,AL
            MOV     AL,8
            OUT     21H,AL
            IDELAY
            MOV     AL,2
            OUT     0A1H,AL
            MOV     AL,4
            OUT     21H,AL
            IDELAY
            MOV     AL,1
            OUT     0A1H,AL
            OUT     21H,AL
            IDELAY
            MOV     AL,INTMAT
            OUT     0A1H,AL
            MOV     AL,INTM
            OUT     21H,AL
ELSE
            MOV     AL,13H
            OUT     20H,AL
            MOV     AL,8
            OUT     21H,AL
            INC     AL
            OUT     21H,AL
            MOV     AL,INTM
            OUT     21H,AL
ENDIF
; clean up to go back to DOS
            LSS     SP,DWORD PTR SOFFSAV
            STI                 ; resume interupt handling
; turn a20 back off
            XOR     AL,AL
            CALL    SETA20
; restore keyboard interrupt
            MOV     DX,KEYOFF
            MOV     AX,KEYSEG
            PUSH    DS
            MOV     DS,AX
            MOV     AX,2509H
            INT     21H
            POP     DS
            MOV     AH,4CH      ; blow this joint!
            MOV     AL,BL       ; get return code
            INT     21H         ; return to the planet of MSDOS
START       ENDP

; Routine to control A20 line. AL=1 to turn A20 on (enable). AL=0 to turn A20
; off (disable). Returns ZF=1 if error; AX destroyed
IF          ATCLASS
SETA20      PROC    NEAR
            PUSH    CX
            MOV     AH,0DFH     ; A20 On
            OR      AL,AL
            JNZ     SHORT A20WAIT1
            MOV     AH,0DDH     ; A20 Off
A20WAIT1:
            CALL    KEYWAIT
            JZ      SHORT A20ERR
            MOV     AL,0D1H
            OUT     64H,AL
            CALL    KEYWAIT
            MOV     AL,AH
            OUT     60H,AL
            CALL    KEYWAIT
            JZ      SHORT A20ERR
            MOV     AL,0FFH
            OUT     64H,AL
            CALL    KEYWAIT
A20ERR:     POP     CX
            RET
SETA20      ENDP

; Wait for keyboard controller ready. Returns ZF=1 if timeout; destroys CX &
AL
KEYWAIT     PROC    NEAR
            XOR     CX,CX       ; maximum time out
KWAITLP:
            DEC     CX
            JZ      SHORT KEYEXIT
            IN      AL,64H
            AND     AL,2
            JNZ     KWAITLP
KEYEXIT:    OR      CX,CX
            RET
KEYWAIT     ENDP

ELSE
; INBOARD PC Code for A20
SETA20      PROC    NEAR
            OR      AL,AL
            MOV     AL,0DFH
            JNZ     A20SET
            MOV     AL,0DDH
A20SET:     OUT     60H,AL
            OR      AL,AL       ; make sure ZF is set for
            RET                 ; compatibilty with AT routines
SETA20      ENDP
ENDIF

; This routine makes a descriptor; ebx=base; ecx=limit in bytes; es:edx=GDT
; address; al= size (0=16bit 1=32bit); ah=AR byte; SI=descriptor (TI & DPL
; not important!). Auto sets and calculates G and limit
MAKE_DESC   PROC    NEAR
            PUSHAD
            MOVZX   ESI,SI
            SHR     SI,3        ; adjust to slot #
            SHL     AL,6        ; shift size to right bit position
            CMP     ECX,0FFFFFH ; see if you need to set G bit
            JBE     SHORT OKLIMR
            SHR     ECX,12      ; div by 4096
            OR      AL,80H      ; set G bit
OKLIMR:     MOV     ES:[EDX+ESI*8],CX
            SHR     ECX,16
            OR      CL,AL
            MOV     ES:[EDX+ESI*8+6],CL
            MOV     ES:[EDX+ESI*8+2],BX
            SHR     EBX,16
            MOV     ES:[EDX+ESI*8+4],BL
            MOV     ES:[EDX+ESI*8+5],AH
            MOV     ES:[EDX+ESI*8+7],BH
            POPAD
            RET
MAKE_DESC   ENDP

; This is the routine that disables ^C interrupts. You could place your own
; code here if desired. NOTE: THIS IS VM86 CODE!
CTRLC       PROC    FAR
            PUSH    DS
            PUSH    AX
            MOV     AX,DAT32
            MOV     DS,AX
            ASSUME  DS:DAT32
            MOV     AL,1
            MOV     BREAKKEY,AL ; set flag
            POP     AX
            POP     DS
            IRET
CTRLC       ENDP
; Reboot handler (VM86 code)
REBOOT      PROC    FAR
            STI
            PUSH    AX
            IN      AL,60H
            CMP     AL,53H      ; delete key?
            JNZ     SHORT NOREBOOT
            XOR     AX,AX
            PUSH    DS
            MOV     DS,AX
            MOV     AL,DS:[417H] ; get shift status
            POP     DS
            TEST    AL,8        ; check for cntl/alt
            JZ      SHORT NOREBOOT
            TEST    AL,4
            JZ      SHORT NOREBOOT
; If detected a ^ALT-DEL then eat it and return
            IN      AL,61H
            MOV     AH,AL
            OR      AL,80H
            OUT     61H,AL
            MOV     AL,AH
            OUT     61H,AL
            MOV     AL,20H
            OUT     20H,AL
            POP     AX
            IRET
; not a ^ALT-DEL, resume normal keyboard handler
NOREBOOT:   POP     AX
            JMP     CS:[KEYCHAIN]
REBOOT      ENDP
; Critical error handler (always fail/ignore)
CRITERR     PROC    FAR
            PUSH    DS
            PUSH    DAT32
            POP     DS
            ASSUME  DS:DAT32
            MOV     CRITAX,AX
            MOV     AL,1
            MOV     CRITICAL,AL
            MOV     CRITDI,DI
            MOV     CRITBP,BP
            MOV     CRITSI,SI
IF          DOS     LT 3
            XOR     AL,AL
ELSE
            MOV     AL,3
ENDIF
            POP     DS
            IRET
CRITERR     ENDP

LAST16      EQU     $
CSEG        ENDS
```

End Listings

Listing One *(Text begins on page 86.)*

```
/* ROLO.H */

/* Menu Constants */
#define SEARCH          2000

#define PRIMARY         2011  /* Index popup */
#define ALTERNATE       2012

#define NEXT            2020
#define PREV            2030
#define INSERT          2040
#define UPDATE          2050
#define DELETE          2060
!!

/* Dialog Constants */
#define PHONE           101
#define PERSON          102
#define COMPANY         103
!!
```

End Listing One

Listing Two

```
/* Additional methods required by WinTrieve Rolodex Browser sample app. */

now(String);!!

/* Returns a copy of the receiver from
  which a blank characters have been removed. */
Def removeBlanks(self ¦ str)
{ str := "";
  do(self,
  {using(c) if c <> ' ' then str := str + asString(c) endif;
  });
  ^str;
}
!!
now(class(CType))!!

/* Return a CType object by looking up it's name in type dictionaries. Same as
   findType method except does no error checking. */
Def getType(self, tName)
{ ^$CTypes[tName] cor $UserTypes[tName];
}
!!
```

End Listing Two

Listing Three

```
/* Rolodex browser. */!!

inherit(Application, #RoloApp, nil, 2, nil)!!

now(class(RoloApp))!!

/* Remove unnecessary classes. */
Def removeExtra(self)
{ do(#(EditWindow WinEllipse WinPolygon StopWatch
       Control FileDialog ReplaceDialog RelFile),
  {using(g) removeGlobal(self, g);
  });
}
!!

now(RoloApp)!!

/* Startup the Rolodex browser. */
Def init(self, cmdStr)
{ init(self:ancestor, cmdStr);
  mainWindow := newMain(RoloWindow, nil, "Rolodex Browser", nil);
  show(mainWindow, CmdShow);
  if not(createDB(mainWindow))
  then close(mainWindow);
  endif;
}
!!

/* Class Initialization */
```

End Listing Three

Listing Four

```
/* A Rolodex file. */!!

inherit(IsamFile, #RoloFile, nil, 2, nil)!!

now(RoloFileClass)!!

now(RoloFile)!!

/* Init Rolodex file record type and key defs. */
Def init(self)
{ init(self:ancestor);
  def(UserType, #rolo, #(
    char    phone       30
    char    person      30
    char    company     30
  ));
  setRecType(self, #rolo);
  addKeyDef(self, #primary, #NODUPS, #phone);
  addKeyDef(self, #person, #DUPS, #person);
}!!
```

End Listing Four

Listing Five

```
/* Main window of Rolodex ISAM file browser. */!!

inherit(TextWindow, #RoloWindow, #(keysDict  /* Dictionary of keys */
roloDB    /* ISAM manager */
roloTable /* ISAM rolodex file */), 2, nil)!!

now(class(RoloWindow))!!

now(RoloWindow)!!

/* Initiate a session with the ISAM manager. Create the ISAM file. */
Def createDB(self)
{ roloDB := new(IsamManager);
  openManager(roloDB);
  if checkError(roloDB)
  then destroy(roloDB);
    ^roloDB := nil;
  endif;
  roloTable := new(RoloFile);
  setFilename(roloTable, "rolo");
  setManager(roloTable, roloDB);
  create(roloTable, ISINOUT + ISMANULOCK);
  if checkError(roloTable)
  then destroy(roloTable);
    ^roloTable := nil;
  endif;
}
!!
/*  Handles input dialog processing for obtaining search title. Returns title
   or nil if user cancels. */
Def getPerson(self ¦ id title)
{ id := new(InputDialog, "Person Key", "Person:", "");
  loop
  while runModal(id, INPUT_BOX, self) = IDOK
    title := leftJustify(getText(id));
    if size(title) > 0
    then ^title;
    endif;
  endLoop;
  ^nil;
}
!!
/* Handles input dialog processing for obtaining phone number. Returns phone
   number or nil if user cancels. */
Def getPhone(self ¦ id str val)
{ id := new(InputDialog, "Primary Key", "Phone:", "");
  loop
  while runModal(id, INPUT_BOX, self) = IDOK
    str := removeBlanks(getText(id));
    if size(str) > 0 cand (val := asInt(str, 10))
    then ^val;
    endif;
  endLoop;
  ^nil;
}
!!
/* Validate record dialog input. If ok, returns dictionary of input field
  values. If error input field, displays error box and returns nil. */
Def validateInput(self, cVals ¦ input)
{ input := new(Dictionary, 10); /* Validate input. */
  input[#phone] := leftJustify(removeBlanks(cVals[PHONE]));
  if size(input[#phone]) = 0
  then errorBox(caption, "Invalid Phone field.");
  else input[#person] := leftJustify(cVals[PERSON]);
    if size(input[#person]) = 0
    then errorBox(caption, "Invalid Person field.");
    else input[#company] := leftJustify(cVals[COMPANY]);
      if size(input[#company]) = 0
      then errorBox(caption, "Invalid Company field.");
      else ^input
      endif;
    endif;
  endif;
  ^nil;
} !!
/* Close the database. */
Def closeDB(self)
{ if roloTable
  then close(roloTable);
    destroy(roloTable);
    roloTable := nil;
  endif;
  if roloDB
  then closeManager(roloDB);
    destroy(roloDB);
    roloDB := nil;
  endif;
}
!!
/* Closing the window so close the database. */
Def shouldClose(self)
{ closeDB(self);
}
!!
/* Initiate a session with the ISAM manager. */
Def openDB(self)
{ roloDB := new(IsamManager);
  openManager(roloDB);
  if checkError(roloDB)
  then destroy(roloDB);
    ^roloDB := nil;
  endif;
  roloTable := new(RoloFile);
  setFilename(roloTable, "rolo");
```

(continued on page 134)

Listing Five *(Listing continued, text begins on page 86.)*

```
  setManager(roloTable, roloDB);
  open(roloTable, ISINOUT + ISMANULOCK);
  if checkError(roloTable)
  then destroy(roloTable);
    ^roloTable := nil;
  endif;
}
!!
/* Display the current record in the window. */
Def printRecord(self)
{ cls(self);
  printString(self, "Phone:     ");
  printString(self, asString(getField(roloTable, #phone)));
  eol(self);
  printString(self, "Person:    ");
  printString(self, asString(getField(roloTable, #person)));
  eol(self);
  printString(self, "Company:   ");
  printString(self, asString(getField(roloTable, #company)));
  eol(self);
}
!!
/* Build a record dialog. */
Def recDlg(self, dPhone, dPerson, dCompany ¦ D)
{ D := new(DialogDesign);
  setSize(D, 8@8, 185@120);
  /*
  addItem(D, newStatic(DlgItem, "Phone:", 100, 5@10, 40@10, 0));
  addItem(D, newEdit(DlgItem, dPhone, PHONE, 50@10, 35@12, 0));
  addItem(D, newStatic(DlgItem, "Person:", 100, 5@25, 40@10, 0));
  addItem(D, newEdit(DlgItem, dPerson, PERSON, 50@25, 125@12, 0));
  addItem(D, newStatic(DlgItem, "Company:", 100, 5@40, 40@10, 0));
  addItem(D, newEdit(DlgItem, dCompany, COMPANY, 50@40, 35@12, 0));
  */
  addItem(D, newStatic(DlgItem, "Phone:", 100, 5@10, 40@10, 0));
  addItem(D, newEdit(DlgItem, dPhone, PHONE, 50@10, 125@12, 0));
  addItem(D, newStatic(DlgItem, "Person:", 100, 5@25, 40@10, 0));
  addItem(D, newEdit(DlgItem, dPerson, PERSON, 50@25, 125@12, 0));
  addItem(D, newStatic(DlgItem, "Company:", 100, 5@40, 40@10, 0));
  addItem(D, newEdit(DlgItem, dCompany, COMPANY, 50@40, 125@12, 0));
  addItem(D, newButton(DlgItem, "Cancel", IDCANCEL, 115@95, 40@14, 0));
  ^D;
}
!!
/* Search for a record based on current index order. */
Def searchRec(self, wp ¦ val)
{ select
    case currentIndex(roloTable) = #primary
    is val := getPhone(self);
      if not(val)
      then ^nil;
      endif;
      putField(roloTable, val, #phone);
    endCase
    case currentIndex(roloTable) = #person
    is val := getPerson(self);
      if not(val)
      then ^nil;
      endif;
      putField(roloTable, val, #person);
    endCase
  endSelect;
  if not(read(roloTable, ISEQUAL))
  then checkError(roloTable);
    ^nil;
  endif;
  printRecord(self);
}
!!
/* Read the previous record and display it. */
Def prevRec(self, wp)
{ if read(roloTable, ISPREV)
  then printRecord(self);
  else checkError(roloTable);
    cls(self);
  endif;
}
!!
/* Read the next record and display it. */
Def nextRec(self, wp)
{ if read(roloTable, ISNEXT)
  then printRecord(self);
  else checkError(roloTable);
    cls(self);
  endif;
}
!!
/* Insert a record. */
Def insertRec(self, wp ¦ rDlg cVals iVals)
{ rDlg := recDlg(self, "", "", "");
  loop
  while runModal(rDlg, nil, self) <> 0
    cVals := controlValues(rDlg);
    if iVals := validateInput(self, cVals)
    then putField(roloTable, iVals[#phone], #phone);
      putField(roloTable, iVals[#person], #person);
      putField(roloTable, iVals[#company], #company);
      if not(insertCurrent(roloTable))
      then checkError(roloTable);
      else printRecord(self);
      endif;
      ^self;
    endif;
    rDlg := recDlg(self, cVals[PHONE], cVals[PERSON], cVals[COMPANY]);
  endLoop;
}!!
```

```
/* Change to selected index and display first record in new index ordering. */
Def changeIndex(self, wp ¦ key oldKey)
{ key := keyAt(keysDict, wp);
  oldKey := currentIndex(roloTable);
  if key = oldKey
  then ^self;
  endif;
  if selectRecord(roloTable, key, 0, ISFIRST)
  then read(roloTable, ISNEXT);
    unCheckMenuItem(menu, keysDict[oldKey]);
    checkMenuItem(menu, wp);
  endif;
  if not(checkError(roloTable))
  then printRecord(self);
  endif;
}
!!
/* Delete the current record. */
Def deleteRec(self, wp)
{ if not(deleteCurrent(roloTable))
  then checkError(roloTable);
  else cls(self);
  endif;
}
!!
/* Update a record. */
Def updateRec(self, wp ¦ rDlg cVals iVals)
{ if not(read(roloTable, ISCURR))
  then checkError(roloTable);
    ^self;
  endif;
  rDlg := recDlg(self,
    asString(getField(roloTable, #phone)),
    getField(roloTable, #person),
    asString(getField(roloTable, #company)));
  loop
  while runModal(rDlg, nil, self) <> 0
    cVals := controlValues(rDlg);
    if iVals := validateInput(self, cVals)
    then putField(roloTable, iVals[#phone], #phone);
      putField(roloTable, iVals[#person], #person);
      putField(roloTable, iVals[#company], #company);
      if not(updateCurrent(roloTable))
      then checkError(roloTable);
      else printRecord(self);
      endif;
      ^self;
    endif;
    rDlg := recDlg(self, cVals[PHONE], cVals[PERSON], cVals[COMPANY]);
  endLoop;
}!!
/* Respond to the menu events.
   The wp argument gives the selected menu ID.
   Get a message symbol from the menu object. */
Def command(self, wp, lp ¦ msg)
{ if msg := action(menu, wp)
  then ^perform(self, wp, msg)
  endif;
}!!
/* Setup the menu bar. */
Def createMenu(self)
{ createMenu(self:ancestor);
  menu := init(new(Menu));
  setHandle(menu, hMenu);
  topMenu(menu, "&Search!", SEARCH, #searchRec);
  keysDict := new(OrderedDictionary, 4);
  keysDict[#primary] := PRIMARY;
  keysDict[#person] := ALTERNATE;
  popupMenu(menu, "&Index",
    tuple("phone", "person"),
    tuple(PRIMARY, ALTERNATE), #changeIndex);
  checkMenuItem(menu, PRIMARY);
  topMenu(menu, "&Next!", NEXT, #nextRec);
  topMenu(menu, "&Prev!", PREV, #prevRec);
  topMenu(menu, "&Insert!", INSERT, #insertRec);
  topMenu(menu, "&Update!", UPDATE, #updateRec);
  topMenu(menu, "&Delete!", DELETE, #deleteRec);
  drawMenu(self);
}
!!
/* Initialize the window. Create menu and About to control menu. */
Def init(self)
{ init(self:ancestor);
  createMenu(self);
  addAbout(self);
}
!!

/* Class Initialization */
```

End Listings

Listing One *(Text begins on page 92.)*

```
/*******************************************************
 * FILE: chekmate.ddl
 * DESCRIPTION: db_Vista database definition language schema
 *******************************************************/

database chekmate {
/* File definition section */
    data file checks1 = "chekmate.d01" contains system, payor, payee,
                                               tranPayee, account;
    data file checks2 = "chekmate.d02" contains transaction, actual;
    data file checks3 = "chekmate.d03" contains budget, distribution;

    key file checkKey = "chekmate.k01" contains strPayorName, strAccountName,
                                               strAccountCode, strPayeeCode,
                                               strPayeeName;
 /* Ewcord definition section   */
    record  payor {
        unique key  char    strPayorName    [35];
                    char    strPayorAddr1   [35];/* single element fields */
                    char    strPayorAddr2   [35];/* are easier in toolbook */
                    char    strPayorCity    [20];
                    char    strPayorState   [3];
                    char    strPayorZip     [10];
                    char    strPayorCtry    [10];
                    char    strPayorTel     [15];
                    char    cPayorRegAdd;   /* values are (a)D(d) and */
                    char    cPayorAccAdd;  /*(a)S(k) on exception processing */
                    char    cPayorPayeeAdd;
                    char    cPayorAccess;  /* values are C(ode) and N(ame) */
                    int     nPayorYear;
                    db_addr dbaPayorCurrBank;
                    int     nPayorNextRecur;
    }

    /*  Account covers both balance sheet and income statement accounts. */
    record  account {
              key  char strAccountName    [35];
              key  char strAccountCode    [15];
                   char strAccountNumber [15];
                   char cAccountType;           /* A L E I E */
                   int  bAccountTax;            /* T= tax related */
                   int  bAccountIsRegister;
    }
    /*  Payee */
    record  payee       {
              key  char    strPayeeName  [35];
              key  char    strPayeeCode  [15];
                   char    strPayeeAddr1 [35];
                   char    strPayeeAddr2 [35];
                   char    strPayeeCity  [20];
                   char    strPayeeState [3];
                   char    strPayeeZip   [10];
                   char    strPayeeCtry  [10];
                   char    strPayeeTel   [15];
                   /* default account is a set */
                   char    strPayeeMemo  [35];
                   char    strPayeeType  [10];
    }
    /*  Budget records are owned by the account. */
    record  budget {
                   int     nBudgetMonth;       /* number from 1 to 12 */
                   long    lBudgetAmount;
    }
    /*  Actuals are maintained in order; speeds process of updating register.*/
    record  actual {
                   int     nActualMonth;
                   long    lActualAmount;
    }
    /*  Transactions are headers for distributions. */
    record  transaction {
                   char    cTranType;          /* check, deposit, etc */
                   char    strTranMemo [35];
                   int     bTranClear;         /* T = cleared */
                   int     nTranNumber;        /* Check number */
                   long    lTranDate;          /* Transaction date as a
                                                  "COBOL" date: ccyymmdd   */
                   long    lTranAmount;
    }
    /*  Distributions are the balancing entries for transactions. */
    record  distribution {
                   long    lDistrAmount;
    }
    /* Set definition section -- Sets with system as parent */
    set system_payor    {
            order   ascending;
            owner   system;
            member  payor by strPayorName;
    }
    set system_payeecode    {
            order   ascending;
            owner   system;
            member  payee by strPayeeCode;
    }
    set system_payeename    {
            order   ascending;
            owner   system;
            member  payee by strPayeeName;
    }
    set system_lastData {
            order   next;
            owner   system;
            member  lastData;
    }
    /* Sets with payor as parent    */
    set payor_accountcode   {
            order   ascending;
            owner   payor;
            member  account by strAccountCode;
    }
    set payor_accounttypecode   {
            order   ascending;
            owner   payor;
            member  account by cAccountType, strAccountCode;
    }
    set payor_accountname   {
            order   ascending;
            owner   payor;
            member  account by strAccountName;
    }
    /* Sets with account as parent */
    set account_budget {
            order   ascending;
            owner   account;
            member  budget      by nBudgetMonth;
    }
    set account_actual {
            order   ascending;
            owner   account;
            member  actual      by nActualMonth;
    }
    set account_distribution {
            order   next;
            owner   account;
            member  distribution;
    }
    /*  account_payee implements default distribution account for payees. */
    set account_payee   {
            order   next;
            owner   account;
            member  payee;
    }
    /*  sets with payee as owner */
    set payee_transaction   {
            order   next;
            owner   payee;
            member  transaction;
    }
    /*  sets with actual as owner. */
    set actual_transaction_date  {
            order   ascending;
            owner   actual;
            member  transaction     by lTranDate;
    }
    set actual_transaction_number   {
            order   ascending;
            owner   actual;
            member  transaction     by nTranNumber;
    }
    set actual_transaction_type {
            order   ascending;
            owner   actual;
            member  transaction     by cTranType, lTranDate;
    }
    set actual_distribution {
            order   next;
            owner   actual;
            member  distribution;
    }
    /*  sets with transaction as owner  */
    set transaction_distribution    {
            order   next;
            owner   transaction;
            member  distribution;
    }
    set transaction_tranPayee   {
            order   next;
            owner   transaction;
            member  tranPayee;
    }

}
```

End Listing One

Listing Two

```
to handle LINKDLLS

        -- use the Windows kernel for memory management
    linkdll "kernel.exe"
        word globalAlloc (word,dword)
        word globalFree (word)
        pointer globalLock (word)
        word globalReAlloc(word,dword,word)
        dword globalSize(word)
        word globalUnlock(word)
    end
        --use ToolBook's file dll for file system access
    linkdll "tbkfile.dll"
        string getCurrentDirectory(string)
        string getCurrentDrive()
    end

        -- use Raima's db_VISTA dll for database functionality
    linkdll "vista.dll"
        --except for close,closetask, and opentask, functions are as
        --in the db_vista docs (dt_ = d_). The last args are *currenttask,dbn

        int dt_close    (pointer)
        int dt_closetask(pointer)
        int dt_connect  (int,pointer,int)
        int dt_crget    (pointer,pointer,int)

        --other calls ommitted here
```

(continued on page 138)

Listing Two *(Listing continued, text begins on page 92.)*

```
        int dt_setro    (int,pointer,int)
    end

        --chekmate.dll is the helper dll for the checking account system
    linkdll "chekmate.dll"
        --other calls ommitted here
        word dbv_EditRegister (word,int,long,int,word,int)
        word dbv_GetRegister (word,int,word)
    end

end
```

End Listing Two

Listing Three

```
to handle InitGlobals
    system svhControl,  svPtrControl
    system svhDBBuffer, svPtrDBBuffer
    system svhCurrTask, svPtrCurrTask
    set svhControl to  GlobalAlloc (66,64) --GHND is 66, size is 64 bytes
    set svhDBBuffer to GlobalAlloc (66,256)
    set svhCurrTask to GlobalAlloc (66,64)
    if svhControl <= 0 or svhDBBuffer <= 0 or svhCurrTask <= 0
        --clean up, since allocation failed
        send FreeGlobals
        send Exit  -- shutdown the system
    else
        --get pointers
        set svPtrControl  to GlobalLock (svhControl)
        set svPtrDBBuffer to GlobalLock (svhDBBuffer)
        set svPtrCurrTask to GlobalLock (svhCurrTask)
    end
end
to handle FreeGlobals
    system svhControl,  svPtrControl
    system svhDBBuffer, svPtrDBBuffer
    system svhCurrTask, svPtrCurrTask
    if svhControl is not null and svhControl > 0
        get GlobalUnlock (svhControl)
        get GlobalFree   (svhControl)
    end
    if svhDBBuffer is not null and svhDBBuffer > 0
        get GlobalUnlock (svhDBBuffer)
        get GlobalFree   (svhDBBuffer)
    end
    if svhCurrTask is not null and svhCurrTask > 0
        get GlobalUnlock (svhCurrTask)
        get GlobalFree   (svhCurrTask)
    end
    --clean up globals to make ToolBook suspend rather than GP Fault
    set svhControl      to null
    set svhDBBuffer     to null
    set svhCurrTask     to null
    set svPtrControl    to null
    set svPtrDBBuffer   to null
    set svPtrCurrTAsk   to null
end
```

End Listing Three

Listing Four

```
typedef struct Control {
   HANDLE  hCurrentTask;        //current task structure (DB_TASK)
   HANDLE  hDataBaseBuffer;     //db_Vista's control buffer

   // Offscreen image / database address pairs
   HANDLE  hRegisterImage;      //offscreen image of the register field
   HANDLE  hRegisterDBAArray;   //array of database addresses, 1 for each line
                                //in the register image
   // Selector fields storage
   HANDLE  hPayeeCodeArray;     //offscreen image of the payee selector field
   HANDLE  hPayeeCodeDBAArray;  //array of database addresses - payee selector
   HANDLE  hPayeeNameArray;     //offscreen image of the payee selector field
   HANDLE  hPayeeNameDBAArray;  //array of database addresses - payee selector
   HANDLE  hAccountCodeArray;   //offscreen image of account selector field
   HANDLE  hAccountCodeDBAArray;//array of database addresses - accts. selector
   HANDLE  hAccountNameArray;   //offscreen image of account selector field
   HANDLE  hAccountNameDBAArray;//array of database addresses - accts selector
   HANDLE  hRegisterNumbers;    //array of deposit, payments, balances

   // Database addresses for active records
   DB_ADDR   dbaCurrPayor;     //address of the current payor record
   DB_ADDR   dbaCurrRegister;  //address of the current active register record
   DB_ADDR   dbaCurrRegisterActual; //address of current month's register

   // Database number for concurrency operations
   int   nDatabaseNumber;      //normally 0
   } CONTROL;
```

End Listing Four

Listing Five

```
--------------------------
--CONTROL BLOCK ACCESSOR--
--------------------------
to get hControl fField
    system svPtrControl
    set vOffset to ControlOffset(fField)
    set retval to -1
    conditions
    when char 1 of fField is "h" --handles
        set retval to pointerWord(vOffset,svPtrControl)
    when chars 1 to 3 of fField is "dba" --db_addrs
        set retval to pointerlong(vOffset,svPtrControl)
    when char 1 of fField is "n"
        set retval to pointerint(vOffset,svPtrControl)
    end
    return retval
end

to set hControl fField to fVal
    system svPtrControl
    set vOffset to ControlOffset(fField)
    conditions
    when char 1 of fField is "h" --handles
        get pointerWord(vOffset,svPtrControl,fVal)
    when chars 1 to 3 of fField is "dba" --db_addrs
        get pointerlong(vOffset,svPtrControl,fVal)
    when char 1 of fField is "n"
        get pointerint(vOffset,svPtrControl,fVal)
    end
end

to get ControlOffset fField
    conditions
    when char 1 of fField is "h" --handles
        conditions
        when fField is hCurrentTask
            set vOffset to 0
        when fField is hDatabaseBuffer
            set vOffset to 1
        when fField is hRegisterImage
            set vOffset to 2
        when fField is hRegisterDBAArray
            set vOffset to 3
        when fField is hRegisterCodeArray
            set vOffset to 4
        when fField is hRegisterCodeDBAArray
            set vOffset to 5
        when fField is hRegisterNameArray
            set vOffset to 6
        when fField is hRegisterNameDBAArray
            set vOffset to 7
        when fField is hPayeeCodeArray
            set vOffset to 8
        when fField is hPayeeCodeDBAArray
            set vOffset to 9
        when fField is hPayeeNameArray
            set vOffset to 10
        when fField is hPayeeNameDBAArray
            set vOffset to 11
        when fField is hAccountCodeArray
            set vOffset to 12
        when fField is hAccountCodeDBAArray
            set vOffset to 13
        when fField is hAccountNameArray
            set vOffset to 14
        when fField is hAccountNameDBAArray
            set vOffset to 15
        when fField is hRegisterNumbers
            set vOffset to 16
        end
        return vOffset*2
    when chars 1 to 3 of fField is "dba" --db_addrs
        set vbase to 34 -- max(vOffset*2)+2
        conditions
        when fField is dbaCurrPayor
            set vOffset to 0
        when fField is dbaCurrRegister
            set vOffset to 1
        when fField is dbaCurrRegisterActual
            set vOffset to 2
        end
        return vbase+vOffset*4
    when char 1 of fField is "n"
        set vbase to 46 --vbase + dbaentries * 4
        conditions
        when fField is "nDatabaseNumber"
            set vOffset to 0
        end
        return vbase+vOffset*2
    end
end
```

End Listing Five

Listing Six

```
-- Fill selector loads selector text from global memory into selector field

to handle FILLSELECTOR fComboName,fObjectName
    set vCurdba to dbv_currentrecord()
    set vHText to hControl("h" & fComboName & "Array")
    if vhText = 0
        get createSelectorList (fComboName)
        set vHText to hControl("h" & fComboName & "Array")
    end
    get globalLock (vHText)
    if fObjectName is null
        set fObjectName to fComboName
    end
    set text of (pListID of group fObjectName) to\
        pointerString(0, it)
    get globalUnLock (vHText)
    set dbv_currentrecord() to vCurdba
end
-- createSelectorList loads global memory block with values from appropriate
-- set member fields
to get createSelectorList fArray
```

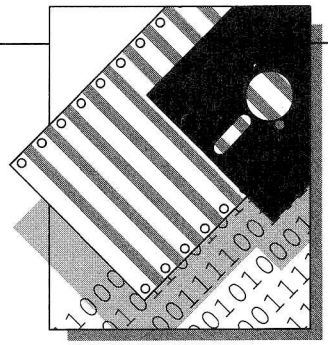
```
    system svPtrDbBuffer, svPtrCurrTask
     -- first determine what all the database constants are
    conditions
    when fArray is "RegisterName"
        get dt_findfm(20000,svPtrCurrTask,0)
        set vSet to 20006
        get dt_setom(vSet,20000,svPtrCurrTask,0)
        set vField to 1000
    -- cases for "RegisterCode", "PayeeCode", "PayeeName", "AccountCode",
    -- and "AccountName" are similar
    else
        return false
    end
    set vArray to "h" & fArray
              --check if the memory is already allocated
              if hControl(vArray & "Array") = 0
        set vhTextBuffer to globalAlloc(66,1000)
        set vhDBABuffer to globalAlloc(66,500)
        set hControl(vArray & "Array") to vhTextBuffer
        set hControl(vArray & "DBAArray") to vhDBABuffer
    else
        set vhTextBuffer to hControl(vArray & "Array")
        set vhDBABuffer to hControl(vArray & "DBAArray")
    end

    set vPtrTextBuffer to globalLock(vhTextBuffer)
    set vPtrDBABuffer to globalLock(vhDBABuffer)
    set vDBAOffset to 0
    set vcharCount to 0
    get dt_findfm(vSet,svPtrCurrTask,0)
    while it = 0
     --build the text buffer
        get dt_crread(vField,svPtrDbBuffer,svPtrCurrTask,0)
        set vline to pointerstring(0,svPtrDbBuffer)
        put CRLF after vLine
        get pointerstring(vCharCount,vPtrTextBuffer,vLine)
        increment vCharCount by charcount(vLine)
      --build the db_addr buffer
        get dt_crget(svPtrDbBuffer,svPtrCurrTask,0)
        get pointerlong(0,svPtrDbBuffer)
        get pointerlong(vDBAOffset,vPtrDBABuffer,it)
        increment vDBAOffset by 4 -- because DBAs are longs
      --get next record
        get dt findnm(vSet,svPtrCurrTask,0)
    end
    return true
    get globalunLock(vhTextBuffer)
    get globalunLock(vhDBABuffer)
end
```

End Listing Six

Listing Seven

```
/***************************************************************************
 * dbv_CreateSelectorList--Purpose: Obtains list of available selections for
 *  a particular chekmate field and save them along with database addresses
 *  in the chekmate control block.
 * Parameters: hControl, HANDLE to the database control block; nSetID, numeric
 *  identifier for set containing selection list; lField, LONG database field
 *  number; nHandleOffset, integer offset into the Control block of handle for
 *  memory where data should be stored.
 * Return Value: 0, if no errors; -n, if errors reading database
 */
extern WORD FAR PASCAL dbv_CreateSelectorList(
      HANDLE   hControl,          // Control Block
      int      nSetID,            // database set identifier
      LONG     lField,            // database field number
      int      hHandleOffset)     // offset in Control Block for memory handle
{
   int      i;
   int            iError=-1;       // error return code
   LPCONTROL      lpControl=NULL;  // control block
   DB_TASK DB_FAR *lpTask=NULL;    // task pointer
   LPHANDLE       lpHandle;        // handle pointer
   if (NULL==hControl)
   {
      return -1;  // control block not initialized
   }

   // Lock control block so task block can be locked for database call.
   lpControl  = (LPCONTROL) GlobalLock (hControl);
   lpTask     = (DB_TASK DB_FAR *) GlobalLock (lpControl->hCurrentTask);

   // point to handle in control block for list text.
   lpHandle   = ((LPHANDLE) lpControl) + hHandleOffset;
   iError = LoadSelectorList (lpHandle,lpHandle+1,nSetID,lField,lpTask,
                                                  lpControl->nDatabaseNumber);
CleanUp:
   GlobalUnlock (lpControl->hCurrentTask);
   GlobalUnlock (hControl);
   return (iError);
}

/***************************************************************************
 * LoadSelectorList--Purpose: Reads database for specified set and transfer
 *   data from lField into text buffer. Each record a separate text line in
 *   buffer and database addresses for each record will also be saved.
 * Parameters: lphListText, handle to memory for list text; lphListDBA, handle
 *   to memory for list database addresses; nSetID, numeric identifier for
 *   database set containing the selection list; lField, LONG database field
 *   number; lpTask, pointer to database task; nDatabase, database number
 * Return Value: 0, if no errors; -n,  if errors reading database
 */
int PASCAL LoadSelectorList (
        LPHANDLE    lphListText,      // handle to memory for list text
        LPHANDLE    lphListDBA,       // memory handle for list database adr.
        int         nSetID,           // database set ID
        LONG        lField,           // database field number for list text
        DB_TASK DB_FAR *lpTask,       // database task
        int         nDatabase)        // database number
{
   int          iError=-1;    // error return code
   int          nDBA=0;       // number of DBA's
   int          nMaxDBA=0;    // maximum number of DBA's used
   int          nMaxBytes=0;  // maximum bytes allowed
   LPSTR        lpText=NULL;  // current text line
   DB_ADDR FAR *lpDBA=NULL;   // database address value
   HANDLE       hMem;         // handle to reallocated memory block
   DB_ADDR      lCurDBA;      // current database record

   // save the current database record
   dt_crget ((DB_ADDR FAR *)&lCurDBA,lpTask,nDatabase);

   // initialize the text and DBA memory blocks.
   if (*lphListText == NULL)
   {
      *lphListText = GlobalAlloc (DLL_ALLOC,SELECTOR_TEXT_SIZE);
   }
   if (*lphListDBA == NULL)
   {
      *lphListDBA = GlobalAlloc (DLL_ALLOC,(LONG)(SELECTOR_DBA_COUNT*sizeof
                                                              (DB_ADDR)));
   }
   if (*lphListText == NULL || *lphListDBA == NULL)
   {
      goto CleanUp;
   }

   // initial allocations to set the maximum values and lock the memory blocks.
   nMaxDBA   = GlobalSize (*lphListDBA) / sizeof(DB_ADDR);
   nMaxBytes = GlobalSize (*lphListText);

   lpText = GlobalLock (*lphListText);
   lpDBA  = (DB_ADDR FAR *) GlobalLock (*lphListDBA);

   // read the database and fill in the text values
   for (iError = dt_findfm (nSetID,lpTask,nDatabase);
        iError==S_OKAY;
        iError = dt_findnm (nSetID,lpTask,nDatabase))
   {
      if (nMaxBytes<=MIN_SELECTOR_TEXT_SIZE)
      {
         // need to allocate more text memory.
         lpText = NULL;
         GlobalUnlock (*lphListText);
         hMem = GlobalReAlloc (*lphListText,
                               GlobalSize(*lphListText)+SELECTOR_TEXT_SIZE,
                               GMEM_ZEROINIT);
         if (hMem==NULL)
         {
            iError = -2;
            goto CleanUp;  // not enough memory
         }
         *lphListText = hMem;  // new handle
         lpText       = GlobalLock (*lphListText);
         nMaxBytes    = GlobalSize(*lphListText) - lstrlen (lpText);
         lpText      += lstrlen(lpText);
      }
      // read the field contents into the text buffer
      dt_crread(lField,lpText,lpTask,nDatabase);
      lstrcat (lpText,"\r\n");
      lpText += lstrlen(lpText);
      // save the DBA of the record
      if (nDBA >= nMaxDBA)
      {
         // need to allocate more DBA memory.
         lpDBA = NULL;
         GlobalUnlock (*lphListDBA);
         hMem = GlobalReAlloc (*lphListDBA,
                   GlobalSize(*lphListDBA)+SELECTOR_DBA_COUNT*sizeof(DB_ADDR),
                                                              GMEM_ZEROINIT);
         if (hMem==NULL)
         {
            iError = -2;
            goto CleanUp;  // not enough memory
         }
         *lphListDBA = hMem;  // new handle
         lpDBA       = (DB_ADDR DB_FAR *)GlobalLock (*lphListDBA);
         nMaxDBA     = GlobalSize(*lphListDBA) - nDBA;
         lpDBA      += nDBA;
      }
      dt_crget   (lpDBA,lpTask,nDatabase);
      lpDBA++;
      nDBA++;
   }
CleanUp:
   // restore the address of the current record
   dt_crset ((DB_ADDR far *)&lCurDBA,lpTask,nDatabase);
   if (lpText!=NULL)
   {
      GlobalUnlock (*lphListText);
   }
   if (lpDBA!=NULL)
   {
      GlobalUnlock (*lphListDBA);
   }
   if (iError==S_EOS)
   {
      iError = S_OKAY;
   }
   return (iError);
}
```

End Listings



Neural Nets: A Cautionary View

Michael Swaine

The September issue of *Byte* magazine brought together some computer industry experts to discuss, among other things, the staying power of some new or newly popular technologies. The expert opinion was divided on whether or not neural nets are a flash in the pan. With all due respect to the experts, neural nets have already demonstrated their usefulness in several areas. If nothing else, they have a place in multidimensional pattern recognition.

Neural nets are a useful tool for solving certain kinds of problems. This column is about what they are not.

The New Connectionism

One thing neural nets are not is just another programming methodology. Neural nets, along with Parallel Distributed Processing, is part of a movement in cognitive and computer science called the "New Connectionism."

Parallel Distributed Processing (PDP) is on the cognitive side of the fence. According to David Rumelhart and James McClelland, whose book *Parallel Distributed Processing* (MIT Press, 1986) defined the discipline, PDP models "assume that information processing takes place through the interactions of a large number of simple processing elements called units, each sending excitatory and inhibitory signals to other units. In some cases, the units stand for possible hypotheses about such things as the letters in a particular display or the syntactic roles of the words in a particular sentence."

That sounds like a description of neural nets, and in fact neural nets more or less represent the computer science and engineering side of the New Connectionism. Generally, a neural net implementation for, say, picking tanks out of the foliage in grainy photographs, looks like a PDP system, but the goals of the implementors are different. Neural nets are built to get something practical done, rather than to model the mind.

There exists today an intricate weaving between cognitive and computer science. This column attempts to follow some of the threads of that common fabric.

The New Connectionism is a recent revival, with differences, of an old idea. Connectionism in this new form has a strong attraction for many computer and cognitive scientists. Jerry Fodor and Zenon Pylyshin examine both the attractions and the assumptions of the New Connectionism in "Connectionism and Cognitive Architecture: A Critical Analysis" in *Connections and Symbols*, eds. Steven Pinker and Jacques Mehler (MIT Press, 1988). "On the computer science side," they say, "connectionism appeals to theorists who think that serial machines are too weak and must be replaced with radically new parallel machines, while on the biological side it appeals to those who believe that cognition can only be understood if we study it as neuroscience. . . . It also appeals to many young cognitive scientists who view the approach as not only anti-establishment (and therefore desirable) but also rigorous and mathematical."

I intend to present here Fodor and Pylyshin's critique of the New Connectionism. Their critique evaluates the New Connectionism as a model of cognitive architecture; that's their interest, and the approach of philosophy, psychology, and linguistics. Why should this interest us as programmers or computer scientists? Because it is very relevant to understanding the potential of neural nets as a programming tool. If the New Connectionism is fundamentally incapable of modeling cognitive architecture, then neural nets are far less powerful than many neural net proponents believe.

It may not be obvious how close cognitive science and computer science have grown in recent years. Even the competitors of the PDP model in cognitive science on the one hand, and the competitors of neural nets in computer science on the other are the same these days. The Classical models that Fodor and Pylyshin set against the New Connectionism *were derived from the structure of Turing and Von Neumann machines.*

Psychological theory today shapes up largely as a battle between cognitive neural nets and cognitive Turing machines.

Rocks Regarded as Real

Fodor and Pylyshin begin by showing that both Connectionist models and Classical models want to operate at the same level of explanation. This is not a trivial point in their domain. Psychology has a long tradition of reductionism that has spawned several distinct schools.

One such school, Behaviorism, was founded in 1913 in a fiery essay by a relatively unknown young psychologist named John Watson. Watson called for a purely objective science of psychology, jettisoning all the fuzzy-headed introspection of the day. There was a

lot to jettison, and so welcome was his argument that not long after this Watson was elected president of the main association of psychologists. Watson subsequently left academia to become an advertising agency executive, perhaps perceiving better than his followers the true mission of Behaviorism.

Behaviorism's most charismatic modern spokesperson was B.F. Skinner. In his autobigraphy, *The Shaping of a Behaviorist*, Skinner characterized his interest as *radical behaviorism, in which the existence of subjective entities is denied.* Skinner died this year, and just weeks before his death he told a reporter that his greatest regret was that he was not understood by his contemporaries. It's true; psychology has moved away from Behaviorist models, and the focus is now on cognitive models that do take things such as thoughts and ideas and mental representations seriously. The reductionism of the Behaviorist school, which may have served a purpose in bringing some rigor to the field 70-odd years ago, now looks deliberately obtuse to most psychologists.

Another reductionist trend of psychology has focused on neural connections. The mind, it maintains, is to be understood in terms of what neurons do: Psychology is neurology, period. This is also not a powerful force in psychology today, and it is important to realize that the New Connectionist models do not, by and large, subscribe to this view. Rumelhart and McClelland, in particular, say that *new and useful concepts emerge at different levels of organization.*

Fodor and Pylyshin contend that neither the New Connectionist models nor the Classical models are reductionist, and that both want to work at a cognitive level of organization; and they explain what they mean by *cognitive.*

The world, they argue, has a causal structure at many levels of analysis. There is a scientific story to be told about quarks, and there is a scientific story to be told about atoms, and about molecules. There is a legitimate science of geology, which legitimately considers such entities as rocks and tectonic plates. While we certainly hope that all these stories will be consistent, we don't call quantum physics a new theory of geology, and deny to geologists the reality of rocks. Different models of explanation are appropriate at different levels of observation.

Fodor and Pylyshin maintain, convincingly and apparently uncontroversially, that the appropriate level of explanation for any account of cognitive architecture (such as the Connectionist or Classical models) is *the representational states of the organism.*

In other words, symbols.

This is an important point for cognitive science because it defines the goals of these two approaches and gives them common observations to examine. Because it defines techniques, it is just as important but far less contentious in computer science. Everyone would agree that you could make a large system maximally fast and efficient by treating it as one entity and coding in machine language. No one would work this way. Divide-and-conquer is one of the most fundamental paradigms of programming. Large programs need intermediate structure, and we would not generally consider it an improvement to strip the objects out of an object-oriented design, the structure from a structured program, the subroutines from a system.

There exists today an intricate weaving between cognitive and computer science

So Connectionist and Classical cognitive science agree about the desired level of explanation; and developers of neural nets and Turing machines agree about the need for intermediate structures. All disputants agree on the need for symbolic processing, although we haven't yet defined what symbols are and how they are to be processed.

So what is the nature of the disagreement between the Classical and Connectionist approaches, which Fodor and Pylyshin say is serious?

It's Not Who You Know, It's How You Know Them

The difference is in what symbols are and in how the system is allowed to operate on them.

For Classical mental models, semantic content is assigned to expressions.

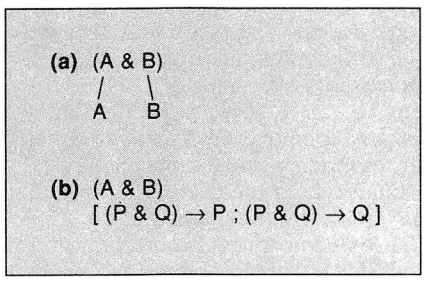


Figure 1: *Connectionist vs. Turing machine*

For Connectionist models, it's assigned to nodes. These are the symbols, the things that represent something, in the two approaches.

The two approaches also differ in the kinds of primitive operations that can be applied to these content-bearing entities. Connectionist models only allow causal connections as primitive relations among the nodes: *When you know how activation and inhibition flow among them, you know everything there is to know about how the nodes in a network are related,* claim Fodor and Pylyshin.

Classical models, on the other hand, allow various relations among their content-bearing entities, including, particularly, the relation of constituency. Here is what that implies: Classical models are committed to what Fodor and Pylyshin call "symbol structures." That is, not all symbols are atomic symbols; some are made up of other symbols. As they put it, some content-bearing entities must have constituents that are also content-bearing, and the content of the composite entity must be a function of the contents of the constituents.

This is crucial to the Classical approach; in particular, it allows the processes of a Classical model to operate on an entity in terms of its structure, so that the same process that converts (P & Q) into P can also convert ((X & Y & Z) & (A & B & C)) into (X & Y & Z).

Consider Figure 1(a) and how a Connectionist machine such as a neural net might interpret it. To the Connectionist machine, the paths in the diagram indicate the possible paths along which excitation and inhibition can flow. When the Connectionist machine draws the inference from (A & B) to A, what happens is that node (A & B) being excited causes node A to be excited.

Now consider a Turing machine drawing the inference from (A & B) to A; see Figure 1(b). The Turing machine contains a program that lets it replace any (P & Q) that it finds on its tape with a corresponding P. It reads (A & B), interprets that as a (P & Q) instance, extracts the P part, which is the A, and puts it on the tape.

Both approaches involve the use of symbols. In the Connectionist machine, the nodes (A & B) and A can represent propositions like *Bill loves Mary and Mary drives a Jeep,* and *Bill loves Mary.* In the Turing machine, the expressions on the tape can represent the same propositions. But the symbols in the Connectionist machine are all atomic, while the symbols in the Turing machine can have structure.

So the architectural difference be-

(continued on page 145)

(continued from page 142)
tween the models is this: *In the Classical machine, the objects to which the content A & B is ascribed literally contain, as proper parts, objects to which the content A is ascribed.* Real-world constituency is modeled in the constituency relations of the Classical machine's objects. But *in the Connectionist machine, none of this is true; the object to which the content A & B is ascribed is causally connected to the object to which the content A is ascribed; but there is no structural (part/whole) relation that holds between them.* Although the label attached to the node (A & B) makes it look like it has structure, it does not.

Here's how Fodor and Pylyshin characterize the disagreement: *Classical and Connectionist theories disagree about the nature of mental representations; for the former, but not for the latter, mental representations characteristically exhibit a combinatorial constituent structure and combinatorial semantics. Classical and Connectionist theories also disagree about the nature of mental processes; for the former, but not for the latter, mental processes are characteristically sensitive to the combinatorial structure of the representations on which they operate.*

Fodor and Pylyshin claim that Connectionist models are wrong on both counts.

A Competency Hearing for Connectionism

Fodor and Pylyshin argue in psychological terms, but there are different ways to test theories and models against human behavior. You can look at actual performance or you can look at competence. Fodor and Pylyshin argue in terms of the latter, in terms of human capacities. The form of the argument is: For any system to be able to do such-and-such a thing that people are able to do, it must have such-and-such a form. There is an analogous argument on the computer science side: Any system that doesn't have such-and-such a form can't do such-and-such interesting things.

Fodor and Pylyshin argue in terms of the productivity of thought, the systematicity and compositionality of cognitive representations, and the systematicity of inference.

Productivity. There is a classic argument, most notably articulated by Noam Chomsky, that purports to prove that certain kinds of mental models can't account for human linguistic competence. Fodor and Pylyshin extend the argument from language to thought. Their argument runs something like this: Human beings are capable of thinking an unbounded variety of thoughts. This unbounded competence must be produced by finite means; there are only a finite number of neurons in the brain. To get unbounded competence by finite means, *you need to treat the system of representations as consisting of expressions belonging to a [recursively] generated set.* This works only when an unbounded number of the expressions are nonatomic. And this is just what can't happen in a Connectionist model. *So,* Fodor and Pylyshin conclude, *the mind cannot be a PDP.*

There is a counterargument to the productivity argument: That humans can't really think infinitely many thoughts. It's unconvincing, but hard to refute. The argument from the systematicity and compositionality of cognitive representations makes it unnecessary to refute it. That argument goes like this:

The ability to think certain thoughts is intrinsically related to the ability to think certain other thoughts. This makes sense only if the thoughts are made up of the same parts. It's not that you couldn't train a neural net to make the right associations between thoughts; it's just that there is nothing in connectionist structure that supports these associations. These associations are crucial; they are the very stuff of which

thought is made, and Connectionist models have no explanation for them.

Finally, there's the argument from the systematicity of inference. A neural net model can be constructed to draw the inference A from (A & B), and to draw the inference B from (A & B). But a neural net can just as easily be constructed to make one of these inferences but not the other. We never see such lopsided mental ability in human beings. Why not? Connectionist models don't know.

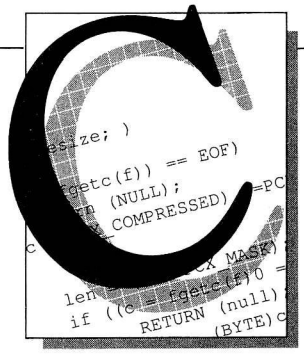
This would seem to imply the following: If you want to build an inference engine that reasons properly, it can't be merely Connectionist.

Fodor and Pylyshin conclude from this sequence of arguments that something is deeply wrong with Connectionist architecture. This is what's wrong: Because Connectionist architecture denies syntactic and semantic structure in mental representations, it is forced to accept, as possible minds, systems that are arbitrarily unsystematic. This is blatantly contrary to observation. Consequently, Connectionist architecture is inadequate to explain the basic data in its domain.

Furthermore, *it's not enough for a Connectionist to agree that all minds are systematic; he must also explain how nature contrives to produce only systematic minds.* That, apparently, the Connectionist can't do without recourse to the only existing approach that does predict pervasive systematicity: The Classical approach.

This seems to me a fairly damning critique of Connectionism as a cognitive architecture, and it seems to have some implications for Connectionist computer programming, as well. For neural nets as programming tools, Fodor and Pylyshin's conclusions would appear to imply at least the following limitation: A neural net, viewed theoretically as a computational system, is the equal of a Turing machine. Both are general-purpose computing machines, and both can achieve anything a general-purpose computing machine can achieve. But for tasks requiring operations on symbol structures, neural nets alone are apparently not enough; for that, you need something like a Turing machine. Neural nets may suffice for the low-level implementation of an inference engine, but only if you use the neural net to implement a Turing machine or other conventional architecture, and implement the inference engine using that.

DDJ
Vote for your favorite feature/article.
Circle Reader Service **No. 11.**



DES Revisited and the Shaft

DATELINE NOVEMBER, 2012: After 22 years of shuffling from court to court, venue to venue, jurisdiction to jurisdiction, the case of FoobarSoft vs. Stevens has finally been settled by the United States Supreme Court. It seems that in the year 1990, the time of the landmark decision where a columnist's opinion was no longer protected from claims of libel, Stevens was a victim of the common columnist's malady known as "pub lag." Unaware of the impending abridgement of the rights of columnists, and working with the columnist's usual four-month lead time, Stevens dutifully reported to his devoted readership that FoobarSoft's new Foobar C++ compiler had a small bug where the Foobar C++ Programmer's Chaise Lounge failed to allow the programmer to assign preferred ratios to the ingredients of the *vodka_martini* class. As a result, programmers were forced to overload, encapsulate, and polymorphize under the influence of two-parts vodka to one-part vermouth. Stevens went on to say that he was unable to maintain mental track of class hierarchies and networks with more than 3500 derived classes while under the influence of instantiated objects of the *vodka_martini* class. He expressed his strong opinion that programmers should be allowed to define drier martinis to taste, and he concluded that FoobarSoft C++ had a deficiency.

After the column was "in the can," as we say, but before the issue was in the hands of readers, the Supreme Court made their ruling. If the opinion of a columnist causes harm to someone and the opinion is not clearly factual, that person has been libeled and is entitled to redress.

Billy Ghoats, FoobarSoft's president, heard from a programmer in Bangor, Maine who said that because of Stevens's remarks, the programmer would delay his paradigm shift until all the compilers got all their ingredients right. Ghoats consulted his legal staff who advised him to sue Stevens for $195.53, the profits that FoobarSoft realize on each sale of the $200 C++ package. Wanting satisfaction as well as compensation, Ghoats also petitioned the court to allow him to scratch "C++" across Stevens's chest as punitive damages.

Now, after 22 years, the case has finally been settled. The many delays were a result of the court's heavy calendar which included the 7,532 lawsuits that were filed against Andy Rooney by food, soap, and cosmetic manufacturers immediately after the 1990 landmark ruling. The decision in the Stevens case was rendered by almost the same Supreme Court that made the original ruling. Chief Justice Wapner, who was appointed by President Quayle in 1998, read the opinion. The court unanimously ruled that Stevens's report of the FoobarSoft C++ behavior did not constitute libel because it was factual. However, calling the behavior a bug and a deficiency was, in the court's opinion, nonfactual and damaging to FoobarSoft, and so the plaintiff received an award in the amount originally requested. The punitive damages were waived because Stevens, now 72 and retired, wears only T-shirts collected over the years from software conferences and vendor-supplied reviewer's packages, and Ghoats was afraid Stevens would insist on wearing the rare and now-collectible "Foobar 'til your pointer drops!" edition given out at the 1995 FoobarSoft-Woodstock Grog and Granola Festival and subsequently banned in Berkeley.

Al Stevens

The indulgence to which I just subjected you is not as far-fetched as it seems. The Supreme Court ruling really happened. We columnists and product reviewers better watch our step. How far can this new attitude reach? If I discuss a compiler, editor, or debugger in this column and do not like it, my opinion becomes lawyer fodder. If the program lacks features, is my opinion that the lack constitutes a deficiency factual or is it libel? Who will be the first to test it?

How far do the implications of this decision reach? Is this opinionated tirade of mine libelous to the Supreme Court itself? Wow! How'd you like to be sued by a bunch of Supreme Court justices?

DES Revisited

In September I published a C implementation of the Data Encryption Standard (DES) algorithm. I built it from the National Bureau of Standards FIPS PUB 46 document which describes the algorithm. I lamented the lack of a validation suite of data that would allow me to determine if the algorithm would comply with the specification. Without one I plodded ahead anyway. The programs seemed to work. They did a dandy job of mangling a file beyond recognition and putting it back in a usable format again.

Richard Feezel, a reader, sent me a message on CompuServe saying that he had compared the output from my programs with that of a hardware board that implemented DES, and the results were different. He spent a good bit of time looking at the code and pointed out some likely problems. Richard did not have the DES document, so he couldn't tell where the code was going astray or if the board and my programs were just different implementations. Later he told me that he found public domain DES code and validation data on CompuServe. The DES hardware he was using passed the tests provided by the validation data.

That was good news. I really needed

a validation data file. The DES code in the download was a bonus that turned out to be a life saver. You will remember that the DES algorithm involves a lot of loops of permutations of bit patterns. An implementor needs a sequence of data blocks that represent the interim steps that the algorithm takes. The input and output examples were helpful but only to tell if the program was working or failing. Nothing really points you to the particular place in the procedure where the first failure occurs. The DES algorithm goes into a loop where halves of a 64-bit data block are permuted, swapped, and mangled with a value derived from the key. The first error, no matter how small, is going to spread like a mushroom.

The big help came from the program that the validation data file accompanied. It is a public domain C program first implemented by James Gillogly and later modified by Phil Karn. You can find it on CompuServe as file DES.ARC in Data Library 3 of the IBMPRO forum. It is no doubt available on other BBSs and online services. The program, which was written some time ago, does not use ANSI C conventions but the authors have included a version that compiles with Turbo C, warnings and all.

By following the progress of the Gillogly/Karn program and comparing its results to those of my program, I was able to correct the problems. Because of the data permutations in DES, I was sure that my implementation of the structure of the algorithm was sound. It was encrypting and decrypting large files without losing any data. The problems had to do with the internal representations of data blocks causing the programs to permute differently than they should. And so they were, most of them. I cleared up most problems by using a *swapbyte* function that changes the byte order of a long integer from aabbccdd to ddccbbaa. The Gillogly/Karn program uses such a function but only in the Turbo C version. Apparently the byte/word architecture of the original machine does not order the bytes in the backward view of 8- and 16-bit machines that has plagued me ever since I first saw a PDP-11.

The DES specification says that the least significant bit of each byte of the key may be for a parity bit or whatever you want it to be

There were a few other problems. I was not rotating the key schedule output correctly. That was a bug that did not prevent the programs from encrypting and decrypting, but the encrypted data blocks were not compatible with the DES specification and probably not as secure.

David Dunthorn of Oak Ridge, Tennessee wrote that my observation about the 8-bit value being preserved in the encrypted file means that my DES code was not working correctly. How true. The behavior I saw was a byproduct of the other problems and led me to a false conclusion. Mr. Dunthorn provided data examples which can serve to test a DES implementation. Example 1 provides three of the examples he sent. The code in this month's column correctly processes these examples as well as that in the downloaded validation data and the other examples from Mr. Dunthorn.

Mr. Dunthorn continues:

> The algorithm for DES is so interconnected that if a program does even a few examples correctly, it has a very high probability of being correct. A one bit change anywhere will yield an entirely different result, not something that is just a little bit off. . . . It is something of a challenge to get DES operating correctly even when working from the specification.

I'll say.

The DES specification says that the least significant bit of each byte of the key may be for a parity bit or whatever you want it to be, and so the algorithm ignores the least significant bit of the key. When my program's output did not match the output from the Gillogly/Karn program, I found that they were using the most significant bit for parity and I was not. The DES specification is not concerned with the implementation problems associated with such things as files and key parity. The specification addresses how you encrypt and decrypt 8-byte blocks of data with an 8-byte key. The parity code in the Gillogly/Karn program is in an outer shell that manages the files and key and calls the inner encryption and decryption code. The inner algorithms comply with the DES specification. From this I realized that different complying DES programs that encrypt and decrypt data files will not always be compatible. I added the odd parity logic to my program so that I could encrypt and decrypt files interchangeably between my program and the downloaded one to test the algorithm.

The documentation that came with the Gillogly/Karn program identified the DES Cipher Block Chaining (CBC) mode, something that is not addressed in the NBS 1977 FIPS PUB DES specification, although the program uses the CBC mode as a default. That had me going for a while. In the CBC mode, the previous ciphered block of 8 bytes is exclusive-orred with the current unencrypted block before the program encrypts the current block. The other mode, which bypasses the extra encryption, is the Electronic Code Book mode. That is the mode supported by my programs. The Gillogly/Karn program implements the CBC mode outside of the DES code, so that is another area where implementations will differ.

I mentioned in September that the DES algorithm has a shortcoming. Because it encrypts blocks of 8 bytes by mangling and scattering the bits all over the block, an encrypted file must necessarily have a length that is a multiple of 8 bytes. The DES algorithm has no way to know the actual length of that last block. In some applications the precise end-of-file location is critical to the use of the file. The Gillogly/Karn program records the length of the last block in the last byte of the file. If the file is an even multiple of 8 bytes, the algorithm adds a block with that information in it. This approach solves the problem, but it is not a part of the DES specification. As with the other external extensions, this solution produces a file format that would not be compatible with other implementations.

Probably the most significant difference between the two implementations is in their execution speed. The Gillogly/Karn program is faster. They do not implement all the permutations by using tables the way I did. Rather, to improve performance, they use certain characteristics of the table values to code some of the permutations with shifts and masks. The cost for those performance improvements is that the code is difficult to understand in places. Often I could not compare their interim results with mine because our approaches were different enough that there were no corresponding data patterns to compare.

The "C Programming" column is as much about C code as it is about algorithms, so I did not attempt to incorporate any of the Gillogly/Karn performance improvements. If I were using DES in a small file environment, such as an electronic mail application, I would probably use my programs because they would be easier for me to understand and maintain. For applications involving large files I would adapt the Gillogly/Karn algorithm to gain its performance benefits. Or I would search out some of the assembly language implementations that are even faster.

I changed the structure of the programs to isolate the DES algorithms from the code that manages the key and the files. You will find the corrected and modified DES programs here in Listing One, des.h (page 164), Listing Two, main.c (page 164), Listing Three, des.c (page 164), and Listing Four, tables.c (page 165). Compile and link everything into an executable pro-

```
Key:  0123456789ABCDEF 0123456789ABCDEF 0123456789ABCDEF
Text: 4E6F772069732074 68652074696D6520 666F7220616C6C20
Encr: 3FA40E8A984D4815 6A271787AB8883F9 893D51EC4B563B53
```

Example 1: *Data examples for DES testing*

gram named "des.exe." Instead of separate encrypt and decrypt programs, such as we had in September, the des program handles both functions. You specify *-e* or *-d* as the first command line parameter to tell the program which operation to perform. The second parameter is the 8-byte key and the third and fourth parameters are the input and output filenames.

The crypto program was like a bicycle lock. It would deter the casual interloper, but the serious thief has the tools and skills to get past it

You can incorporate the DES functions into your own programs by including des.h and linking to des.c and tables.c. Call the *initkey* function with the address of your 8-byte key as an argument. Then call either the *encrypt* or *decrypt* function once for each 8-byte block in the data you want to encrypt or decrypt. Both functions accept a pointer to the 8-byte block where the input to and output from the algorithm will go.

The CRYPTO Program

A reader named Dave called to say that crypto.c, my first example of a simple encryption/decryption algorithm was not very good. It seems that he encrypted a file that had long strings of zero bytes. The crypto program encrypts and decrypts by simply exclusive-orring the 8-byte key with each successive 8-byte block of the data file. When the file has strings of zero bytes, the encrypted value is the key itself.

I designed crypto.c to encrypt ASCII electronic mail message text, which never has strings of zero-value bytes. I should have warned you about that particular behavior, however.

Another reader, Roberto Quijalvo of Plantation, Florida wrote to point out that if your ASCII text file has strings of spaces and the key is also ASCII text, the exclusive-or inserts the key into the text with a case change. That one never bit me because my ASCII files go through a run-length encoder, which truncates trailing white space from lines and compresses other runs of characters into escape sequences prior to encryption. That process was not for encryption, though, but for compression minimize transmission time.

Mr. Quijalvo contributed some code +to internally mangle the key. He says

> A slight alteration in [the] code would make it more difficult to discover the key:
>
> ```
> /* original code segment */
> while (*cp1 && cp1 < argv[1]+8)
>    *cp2++ ^= *cp1++;
>
> /* altered code segment */
> while (*cp1 && cp1 < argv[1]+8)
>    *cp2++ ^=
>       (*cp1<0x41 ? *cp1++ :
>          *cp1++ - 0x40 );
> ```
>
> By checking the key for a character greater than 0x40 ('@') and changing any alpha character to a non-alpha character before using it, the key can be saved from experiencing such embarrassingly blatant exposure.

Francis Rocks of Berwyn, Pennsylvania reported the same problem and offered a different solution. Francis says,

> A solution would be to eliminate hex 20 (blank) encoding with the following code replacing your second "while" construct:
>
> ```
> while (*cp1 && cp1 < argv[1]+8) {
>    if (*cp2 != 0x20) {
>       *cp2++ ^= *cp1++;
>    }
>    else {
>       *cp2++;
>       *cp1++;
>    }
> }
> ```

All three readers thought that crypto.c was less secure than it ought to be, which it is if you use it outside of a controlled data environment such as the one I built the algorithm for.

These modifications are, of course, merely moves to beef up a simple algorithm whose original intent was only to momentarily divert the merely curious, which is the cause for encryption for most of us. In more hostile surroundings if the invader knows that the text is encrypted, he or she must discover the algorithm as well as the key. Because the exclusive-or algorithm is a common one, the appearance of any repeated pattern, even the mangled key, would tend to alert the spy to the possibility of exclusive-orring the entire message with different permutations of that pattern to see what would happen. Rocks's solution works only for the strings of spaces. But if a file has repeated strings of other values, fixed permutations of the key are going to be in there, and a codebreaker will exploit the patterns.

The crypto program was like a bicycle lock. It would deter the casual interloper, but the serious thief has the tools and skills to get past it. If you need to be that secure, you need something other than crypto.

Listing Five, page 166, and Listing Six, page 166, are encrypto.c and decrypto.c, two programs that eliminate the exposed key problems that crypto.c has with text files. I combined the run-length encoding algorithm of my text compresser and the encryption of the crypto.c program into these two programs. Besides encrypting and decrypting, they compress runs of duplicate characters into a simple escape sequence. The programs work only with ASCII text files.

The encrypto.c program translates any run of two or more duplicated characters into 2 bytes. The first byte is the count of the run with the most significant bit set on. The second byte is the duplicated character. Because the algorithm uses the most significant bit to identify a run-length counter, it works only with 7-bit ASCII files and will terminate if it finds a most significant bit set in the input file. After encrypto.c compresses the text, it encrypts it in blocks with lengths equal to that of the key's length. The compression and encryption are done in one pass of the file.

The decrypto.c program reverses the compression/encryption process. It decrypts the characters and then decompresses them.

The crypto.c program in September used a fixed-length key of 8 bytes. encrypto.c and decrypto.c take the key length that is entered and encrypt/decrypt blocks of that length.

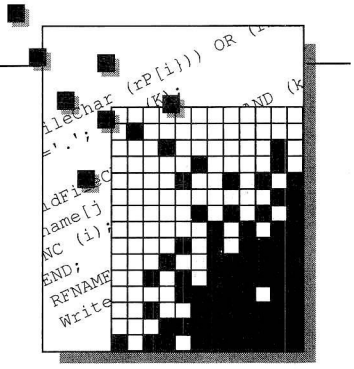
I considered writing an identifying signature at the front of the encrypted file so that the decrypto.c program could verify that it was being asked to decrypt a properly encrypted file. I rejected this idea because an outsider who intercepted the file might then know which program encrypted it. I then considered encrypting the signature. I rejected that idea as well. A codebreaker who already knew the algorithm would then have a known data value from which to reverse-engineer the key.

While not immune to the attacks of a persistent codebreaker, the algorithms in encrypto.c and decrypto.c are more secure than those of the simpler crypto.c program from September and far less complex than those of DES.

DDJ

(Listings begin on page 164.)

Vote for your favorite feature/article.
Circle Reader Service **No. 12.**



Ice Cubes in the Swimming Pool

Jeff Duntemann, KI6RA/7

Be careful what you ask for, children: *You might get it.*

Consider: It's hot out here in North Phoenix, so hot that by August the swimming pools have become too warm to be refreshing. The best you can do is jump in and jump immediately out again in the hope that the single-digit humidity will cool you off by rapid evaporation. If you have a solar pool-heating system, you can cool the water by running it at *night*, so that the hot water warms the panels and radiates heat into the clear desert midnight sky.

No such luck here until we move into our new house, and by then summer will be little more than a memory with smoke wafting off of it. So when I mailed invitations to one of our regular parties, I made it plain to our guests: The pool temperature had gone over 90 degrees, and if it got any hotter I'd have to throw some ice cubes into it.

It was 105 degrees the afternoon of the party, and no one was in the pool. Come seven PM, we had finished dinner and were talking hacker-talk out on the patio. The wind was rising sharply, and we saw the flashes of distant lightning in the north. Low clouds rolled over the sun, and behind them came roiling clouds of red dust from the desert just north of town.

The storm hit like a hammer, taking the temperature from 105 down to 75 almost instantly, with continuous lightning and winds gusting to 70 miles per hour amidst torrential rain. Then, as though that weren't enough, it started to *hail.*

Big chunks of ice riding a 70 mph wind are lethal missiles. We tumbled inside and watched in amazement as the storm poured wave after wave of hailstones, some rivaling golf balls, into the yard. Ten minutes later, there were hailstone drifts three inches deep beside the fence, and a thick layer of hailstones bobbed steaming in our pool.

Out in front, 10th Avenue was a river, carrying broken tree branches and fragments of redwood fence. Power was out for most of the night.

But I had managed to get some ice into the pool.

Modulus *Vobiscum*

Similarly, I got what I asked for — and then some — when I swore up and down that I would figure out what was eating my day-of-the-week function, *CalcDayOfWeek*. Part of that adventure was detailed last month, in my recreation of a Pascal and Modula-2 implementation of Zeller's Congruence from the original source, a scholarly paper that Zeller published in 1887. (Zeller's Congruence, in case you're just tuning in, is an algorithm that calculates the day of the week, given the year, month, and day.)

I left you with an implementation that worked fine. There was a worm in it, though: In order to make the algorithm return correct dates in all cases, I had to ensure that Turbo Pascal's MOD operator never acted upon a negative quantity. In other words, rather than evaluate –17 MOD 7, I added 7 repeatedly to –17 until the sum surfaced on the plus side of zero. That quantity then became the first operand of the MOD operator. (See last month's code listings if you don't quite see what I mean.) This is a kluge by the classic definition, which is to say, something that works right for all the wrong reasons. (And the wrongest reason of all is not having any idea whatsoever why it works right!)

I'd heard whispers for years that Turbo Pascal's X MOD Y operator wasn't quite kosher, but nobody I knew could define just what was wrong with it. All my tests indicated that it did just what the modulus operation was supposed to do: Calculate an integer division of the first operand by the second, then return the remainder of that division. The dividend is thrown away unused. (That's what the X DIV Y operator is for.) No matter what combination of positive and negative integers I used, every calculation agreed with my painstaking pencil-and-paper work. (How many pocket calculators will show you a remainder? Not mine . . . .) I began to consign those rumors of MOD's incompetence to base canards, but then a disturbing possibility turned up out of a conversation in the mathematics conference on BIX: That the way Turbo Pascal and I understood the modulus operator was subtly but ruinously different from the way the mathematics community — and hence Zeller himself — understood it.

Which Way is the Floor?

My head first started to spin when someone known to me only as "seba" posted the following definition for the modulus operator:

```
X MOD Y = X–Y*[X/Y]
```

where *[X/Y]* is the greatest integer less than or equal to *X/Y*. This sure didn't look like taking the remainder and tossing the dividend, and it sure wasn't. I first implemented this equation the following way:

```
X–(Y*Trunc(X/Y))
```

This, again, worked for positive values

of *X* but not for negative values. After considerable head scratching, I realized that truncating a real number value such as that produced by *X/Y* always moves the value toward zero on the number line. This is fine for positive values, but for negative values, moving toward zero makes the value *greater* than *X/Y* rather than less. Implementing the equation for negative values has to be done this way:

```
X- (Y*Trunc((X/Y) -1))
```

Decrementing the value of the expression *X/Y* by one ensures that truncating the expression returns the largest integer *less* than *X/Y*.

Combining the two cases into one Modulus function is easy, and the result is the function given in Listing One (page 167).

Modulus Roulette

If you have yet to grok the fullness of this new equation describing the modulus operator, have faith — I trust it because it works, not because it's "obvious" to me. For it to become obvious, I have to look to a graphical interpretation of the modulus operator, something I learned in eighth grade during a brief brush with the New Math Monster and haven't thought about since 1966. To implement the modulus operator graphically, follow the example shown in *Figure 1*. To take *N* modulus *Y*, draw a circle of digits starting from *0* and running to *Y–1*. In Figure 1, I'm setting up the circle to take *N* modulus *7*, as required by Zeller's Congruence.

With the circle in place, start at 0 and count around the circle by *N* positions. (Tap each digit with a pencil eraser if you have to — I did, and I won't laugh if you do too.) Here's the kicker: If *N* is positive, move around the circle in a clockwise direction. If *N* is negative, move around the circle in a counterclockwise direction. Try it right now with the numbers *17* (clockwise) and *–17* (counterclockwise.) Your answers should be:

I'd heard whispers for years that Turbo Pascal's X MOD Y operator wasn't quite kosher, but nobody I knew could define just what was wrong with it

```
17 modulus 7 = 3
–17 modulus 7 = 4.
```

Note that I didn't call the operator MOD. By now it was clear to me that MOD and modulus are two different operators. Turbo Pascal and Turbo C++ calculate *17* modulus *7* = *3* but *–17* modulus *7* = *–3*. In all cases, *–N MOD Y* is simply *– (N MOD Y)*, which is what you would expect if MOD returns the remainder of a simple division. Unfortunately, what we call MOD Zeller would have called the remainder function. And that in a nutshell was what was wrong with *CalcDayOfWeek*. (The final implementation of *CalcDayOfWeek* using Zeller's Congruence is given in Listing Two, page 167.) It wasn't really a bug in Turbo Pascal. What TP calls MOD is really the remainder function, and if what you need is the remainder function it will work without a hitch every time.

However, if you've ever implemented an algorithm from the literature of mathematics that calls for the modulus operator and used MOD, your algorithm will go wonky any time the first operand of modulus goes negative. Check it out. *Now*. I suspect the problem may be present in many more compilers than Borland's. If you're not using Turbo Pascal, bring up your compiler and evaluate *–17 MOD 7*. If you get anything but 4, you've got remainder or something else, not true modulus.

Thus ends the tale of what is certainly the most amazing single bug in my very full notebook of bug hunts.

BGI Hardcopy

Sometimes you wonder why it takes as long as it does for certain new products to appear. Ever since Borland first released the BGI with Turbo Pascal 4.0, there's been this gaping hole in BGI functionality: hardcopy. While numerous utilities exist to zip a displayed graphics screen out to the printer, this requires that the graphics image first appear on the screen. If you want to use the screen for something else, you're simply out of luck.

Enter Graf/Drive Plus from Fleming Software. The notion is elegant: A BGI driver that sends BGI output to a hardcopy device rather than a video device. In other words, rather than load a driver for the EGA or Hercules graphics adapters, you load a driver for the Epson or LaserJet printers or one of the supported HP plotters. Then, when you execute a standard BGI statement such as *Circle* or *OutTextXY*, the graphics go to paper rather than to the screen.

All BGI functions that make sense for hardcopy are supported. (Those that don't are things such as *GetImage*, *PutImage*, and palette control routines.) The package works with Turbo Pascal (*5.x*), Turbo C 2.0, and Turbo C++ 1.0. Printers currently supported are the HP LaserJet series, Epson MX/FX 80 and compatibles, the Epson LQ1500 and compatibles, and the HP 7470A and 7475A pen plotters. Several new printers will be supported by the time you read this, including PostScript, the HP PaintJet, the HP7585 plotter, and the Toshiba and IBM QuietWriter printers.

Graf/Drive Plus does exactly what it says it does. I've had no trouble making it work and can trace no crashes

(continued on page 159)

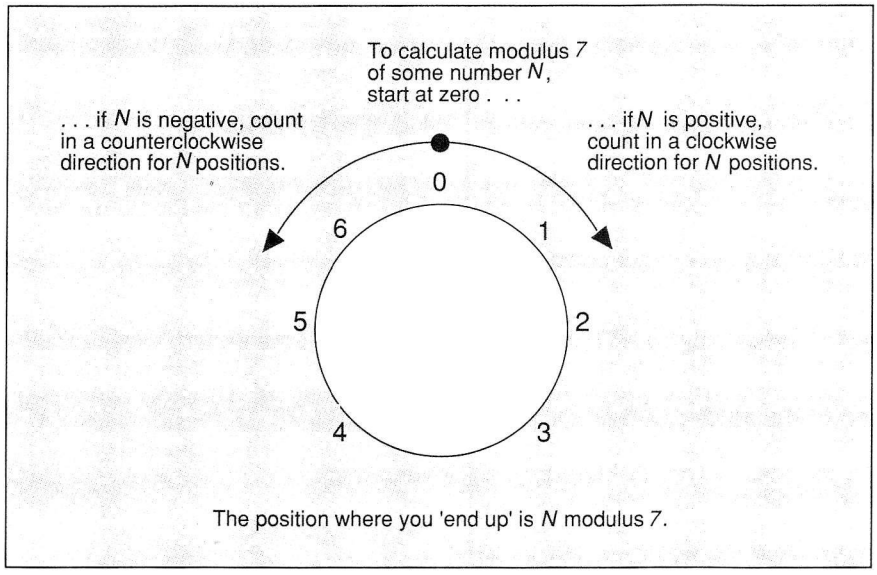


Figure 1: *A graphical interpretation of the modulus operator*

(continued from page 156)
or other bugs to its use. The individual drivers are fairly small (less than 16K) and can be linked into your .EXE image just as any BGI driver can be. The hacking-around license is $149; if you purchase a developers' license ($299) you can distribute the drivers with your application without further royalty obligations.

Lord knows, something like this should have been done years ago. Highly recommended.

Comdex Approaches

A lot of developers don't go to Comdex anymore. They say, rightfully, that it's a hardware show, but that's all the more reason to hock your firstborn and truck on out there. For the little guy, it's a niche-filler's game these days, and you have to remember that a software niche is an idea delimited by available hardware. And the one thing a lone wolf can do better than all the armies of IBM is *have ideas.*

It happens to me every year: I prowl the aisles (especially in the third-tier hotels where the weird stuff ends up) and wait for the hardware on display to suggest new applications to me. I'm in the magazine business rather than the software business these days, but the ideas come nonetheless. And sometimes the triggering products are not even "hardware" in the familiar sense: Last year I saw a firm dealing in custom-perfed sheets of paper for computer printed tear-out coupons, and I had this notion that I could create custom sheets punched and perfed to tear down into pages for the smallest standard pocket memo book, which is something only a little larger than your average business card. Feed 10 or 12 sheets through your laser printer with the proper software interface to your contacts database (including an eight-point font) and bang! A *little* little black book!

The problem of marketing your software still remains (looming larger than ever, in fact) but having ideas that nobody else has had is often no harder than getting out and exposing yourself to large quantities of the stuff of which our industry is made.

Products Mentioned

Graf/Drive Plus
Fleming Software
P.O. Box 528
Oakton, VA 22124
703-591-6451
Personal license: $149
Developers' license: $299

After all, lots of people find that pacing around the room helps shake ideas loose. At Comdex, you can pace around some of the largest rooms in the civilized universe, (in fact, to see it all you'd better pace at something close to a dead run) surrounded by the largest concentration of computer stuff that ever happens anywhere.

If you can't get ideas *there*, you might as well go back to selling shower curtains.

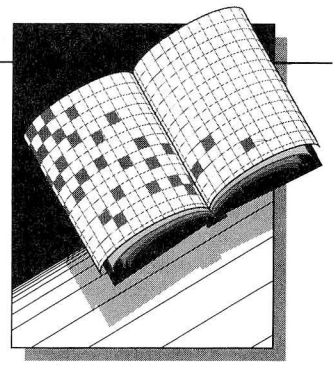
Source Code Availability

As a service to our readers, all source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Drive, Redwood City, CA 94063, or call 800-356-2002 (inside Calif.) or 800-533-4372 (outside Calif.). Specify issue number and disk format. Code is also available through M&T's Telepath online service (via TYMNET) and through the DDJ Forum on CompuServe (type GO DDJ).

DDJ

(Listings begin on page 167.)

Vote for your favorite feature/article.
Circle Reader Service **No. 13.**



Object-Oriented Software Development: Reality Sets In

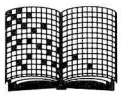
Andrew Schulman

With so many books now available on particular object-oriented programming languages, it is surprising that there aren't more books available to explain what *object-oriented* means. Books on C++ or Smalltalk tell you how to use these languages to implement a system, but they don't tell you much about the object-oriented approach to *designing* that system in the first place.

Those books which have been written on object-oriented design, tend to be one-sided using Fred Brooks' pronouncements to describe the pitfalls and tar pits of traditional software development, but without much understanding of the potential pitfalls that object-oriented design, like any design principle, is bound to have.

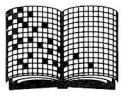
Grady Booch's *Object-Oriented Design with Applications* isn't like this. It has a strong sense of the real-world issues involved with adopting and using the object-oriented paradigm. If you are looking for a magic solution to the problems of software development, you will probably find this book irritating. But for a balanced assessment of the pros and cons of object-oriented software development, this is a fine place to start.

Towards the beginning of the book, Booch says that it is good practice to write programs "so that they don't care about the physical representation of data." How many times have we heard this from the advocates of object-oriented design? How many times *can* you actually write non-trivial programs that are in fact agnostic as to the physical representation of data?



Object-Oriented Design with Applications

Grady Booch
Redwood City, Calif.:
The Benjamin/Cummings
Publishing Co., 1991
580 pp., $38.75
ISBN 0-8053-0091-0.



Object-Oriented Software Construction

Bertrand Meyer
New York, N.Y.: Prentice Hall, 1988
534 pp., $41.00
ISBN 0-13-629049-3.

Well, unlike more amateurish books on object-oriented software development, Booch doesn't sweep these issues under the rug. Chapter 7 (entitled "Pragmatics") is particularly good at describing the potential performance risks of object-oriented design. For example, Booch plainly states that object-oriented software rarely has "locality of reference," so that special measures must be taken for it to behave on systems with paged virtual memory. If we care about performance (!), we often must care about the physical representation of data too. And while object-oriented software can actually give *better* performance, because, for example, virtual functions eliminate the need for some explicit type checking, on the other hand it can often result in code bloat. Thus, Booch points out that sophisticated compilers and linkers with dead-code elimination are needed to fulfill some of the promise of object-oriented software development.

Booch also shows a grave concern with "performance" in software development time. Several passages are devoted to design issues that affect *recompilation*: "A change in a single module interface might result in many hours of recompilation. Obviously, a manager cannot often afford to allow this" (p. 52). "In the extreme, recompilation costs may be so high as to inhibit developers from making changes that are reasonable improvements" (p. 205). Thus, to be practical, object-oriented software development also cries out for incremental compilers.

Booch discusses the key issue of how an organization makes the transition to object-oriented software development. One section describes the "dark side" of start-up costs; Booch makes the reasonable recommendation that one start using object-oriented design in a low-risk project first. The book also discusses possible organizational obstacles: "Object-oriented design

makes it possible to use smaller development teams . . . . Unfortunately, trying to staff a project with fewer people than traditional folklore suggests are needed may produce resistance. Such an approach jeopardizes the attempts of some managers to build empires" (p. 208).

While Booch's understanding that every benefit has an associated cost makes this an excellent introduction to object-oriented software development, I found the actual discussions of *design* less useful. There is a big build-up to the chapter on "Classification": Presented with a problem, how do you find "where the objects are"? Somewhere Booch makes the central point that *not everything is an object*, and that here, as elsewhere, there is no prescription for turning a problem into a neat bundle of classes and objects.

But the chapter on "Classification" was disappointing. Booch writes here that "To the developer in the trenches fighting changing requirements amidst limited resources and tight schedules, our discussion [of classification philosophy, such as Wittgenstein on categories!] may seem to be far removed from the battlefields of reality. Actually, these approaches to classification have direct application to object-oriented design" (p. 140). Unfortunately, the connection is never really made, making this chapter the book's one disappointment.

The entire second half of *Object-Oriented Design With Applications* contains five applications in five languages: A home heating system in Smalltalk, a geometrical optics construction kit in Object Pascal (MacApp), a software house-bug reporting system in C++ (a good example of a useful, yet low-risk project for an organization making the transition to object-oriented development to cut its teeth on), a cryptanalysis program in the Common LISP Object System (CLOS), and a traffic management system in Ada (Booch is well known for his work on Ada).

Execution of an Eiffel system may be viewed as a cooperative race between two coroutines

Booch states that *object-oriented* has emerged as a "unifying theme" in "diverse segments of the computer sciences." Apparently this theme can be useful in *any* area where it is desirable to close the "semantic gap," and he cites several examples of "object-oriented operating systems" and even "object-oriented hardware," including the Intel 432 and its iMAX operating system. On the other hand, the RISC paradigm in computer architecture asserts that there is nothing intrinsically wrong with the "semantic gap;" object-oriented hardware indeed sounds like a bad idea. Still, since I work at a company which produces system software (Phar Lap, makers of 386|DOS-Extender), Booch's book had me wonder about the possible applicability of the object-oriented paradigm to low-level software such as ours. Recently, Microsoft has been talking about "object-oriented operating systems." Frankly, I'm not convinced.

While reading the new book by Grady Booch, I had the pleasure of rereading, for the third time, Bertrand Meyer's *Object-Oriented Software Construction.* This is still the standard work on the topic, and the best explanation of what all the object-oriented fuss is about.

Meyer's book describes the Eiffel lan-

guage, so you might think it is useless if you don't have access to Eiffel. But, by using a programming language to which the reader probably doesn't have access, *Object-Oriented Software Construction* makes you concentrate on design issues, not on the peculiarities of language syntax.

The opening sentence of the book states that "The principal aim of software engineering is to help produce quality software." Yet, "quality in software is best viewed as a tradeoff." Meyer also emphasizes the importance of software maintenance, stating that it is "the hidden part, the side of the profession which is not usually highlighted in programming courses." In fact, an awareness of the importance of maintenance has been one of the driving forces behind the object-oriented movement.

Much of the book is concerned with issues of writing correct and robust software, and of dealing with errors, exceptions, failures, and abnormal conditions. Some of this material appears in "Writing Correct Software with Eiffel," by Bertrand Meyer. (See either *DDJ*, December 1989 or *DDJ* bound volume 14.) A glance at the book's index under the entries "exception" and "failure" will point you to several lengthy discussions of these often-neglected issues. The sections "Coping with Failure" (pp. 144-155) and "Dealing with Abnormal Cases" (pp. 199-203) are superb.

One of Meyer's key techniques for error handling is the assertion, which he refers to a way to "include specification elements within the implementations themselves" (p. 112). Meyer introduces a "class invariants" notation which is, it seems, far more understandable than notations which have appeared in books such as Barbara Liskov and John Guttag's *Abstraction and Specification in Program Development.*

Meyer devotes an entire chapter to object-oriented memory management. The chapter begins with the statement "It would be so nice to forget about memory" (p. 352), but Meyer has no illusions on that score. Meyer shows the problem with the traditional approaches to memory management, including what he calls "programmer-controlled deallocation," which turns the programmer into a bookkeeper. What then characterizes object-oriented memory management? Obviously some form of "automatic" memory management. "Unfortunately, reference counting is not a realistic technique" (p. 365), and garbage collection "is unacceptable for real-time applications" (p. 366).

The Eiffel approach to storage management is essentially a garbage-collection coroutine: "Execution of an Eiffel system may be viewed as a cooperative race between two coroutines: The application, which creates objects as it goes, and renders some dead; and the collector, which chases after the application, collects all the dead objects it can find, and makes them available again to the application" (p. 367).

Throughout, *Object-Oriented Software Construction* provides insights into the necessary object-oriented "mindset." This mindset has a "more neutral attitude toward ordering." "Real systems have no top" (p. 47). Instead, object-oriented systems have a "shopping list approach." Always try to pass the buck: "Better later than sooner, says object-oriented wisdom" (p. 50). Meyer argues that *looser* systems will be more robust, extendible, reusable, and maintainable than current systems: "decentralization is the key to flexible architectures" (p. 431). The object-oriented approach might be caricatured as nothing more than "lighten up, dude." Meyer shows how the looser, bottom-up, topless approach really can result in better quality software.

In short, while Booch's new book is more tempered by knowledge of the costs and possible downsides of object-oriented software development, Meyer's classic work is a better introduction to object-oriented thinking. The two books complement each other well, and are recommended to anyone trying to understand what it means to be object oriented.

DDJ
Vote for your favorite feature/article.
Circle Reader Service **No. 14.**

Listing One *(Text begins on page 149.)*

```
/* -------------- des.h ---------------- */
/* Header file for Data Encryption Standard algorithms  */

/* -------------- prototypes ------------------- */
void initkey(char *key);
void encrypt(char *blk);
void decrypt(char *blk);

/* ----------- tables ------------ */
extern unsigned char Pmask[];
extern unsigned char IPtbl[];
extern unsigned char Etbl[];
extern unsigned char Ptbl[];
extern unsigned char stbl[8][4][16];
extern unsigned char PC1tbl[];
extern unsigned char PC2tbl[];
extern unsigned char ex6[8][2][4];
```

End Listing One

Listing Two

```
/* Data Encryption Standard front end
 * Usage: des [-e -d] keyvalue infile outfile
 */

#include <stdio.h>
#include <string.h>
#include "des.h"

static void setparity(char *key);

void main(int argc, char *argv[])
{
   FILE *fi, *fo;
   char key[9];
   char blk[8];

   if (argc > 4)    {
        strncpy(key, argv[2], 8);
        key[8] = '\0';
        setparity(key);

        initkey(key);
       if ((fi = fopen(argv[3], "rb")) != NULL)    {
           if ((fo = fopen(argv[4], "wb")) != NULL)    {
               while (!feof(fi))    {
                   memset(blk, 0, 8);
                   if (fread(blk, 1, 8, fi) != 0)    {
                       if (stricmp(argv[1], "-e") == 0)
                           encrypt(blk);
                       else
                           decrypt(blk);
                       fwrite(blk, 1, 8, fo);
                   }
               }
               fclose(fo);
           }
           fclose(fi);
       }
   }
   else
       printf("\nUsage: des [-e -d] keyvalue infile outfile");
}

/* -------- make a character odd parity ---------- */
static unsigned char oddparity(unsigned char s)
{
   unsigned char c = s ¦ 0x80;
   while (s)    {
       if (s & 1)
           c ^= 0x80;
       s = (s >> 1) & 0x7f;
   }
   return c;
}

/* ------ make a key odd parity ------- */
void setparity(char *key)
{
   int i;
   for (i = 0; i < 8; i++)
       *(key+i) = oddparity(*(key+i));
}
```

End Listing Two

Listing Three

```
/* ---------------------- des.c --------------------------- */
/* Functions and tables for DES encryption and decryption
 */

#include <stdio.h>
#include <string.h>
#include "des.h"

/* -------- 48-bit key permutation ------- */
struct ks    {
    char ki[6];
};

/* ------- two halves of a 64-bit data block ------- */
struct LR    {
    long L;
    long R;
};

static struct ks keys[16];

static void rotate(unsigned char *c, int n);
static int fourbits(struct ks, int s);
static int sixbits(struct ks, int s);
static void inverse_permute(long *op,long *ip,long *tbl,int n);
static void permute(long *op, long *ip, long *tbl, int n);
static long f(long blk, struct ks ky);
static struct ks KS(int n, char *key);
static void swapbyte(long *l);

/* ---------- initialize the key ------------- */
void initkey(char *key)
{
    int i;
    for (i = 0; i < 16; i++)
        keys[i] = KS(i, key);
}

/* ---------- encrypt an 8-byte block ------------ */
void encrypt(char *blk)
{
   struct LR ip, op;
   long temp;
   int n;

   memcpy(&ip, blk, sizeof(struct LR));
   /* -------- initial permuation -------- */
   permute(&op.L, &ip.L, (long *)IPtbl, 64);
   swapbyte(&op.L);
   swapbyte(&op.R);
   /* ------ swap and key iterations ----- */
   for (n = 0; n < 16; n++)    {
       temp = op.R;
       op.R = op.L ^ f(op.R, keys[n]);
       op.L = temp;
   }
   ip.R = op.L;
   ip.L = op.R;
   swapbyte(&ip.L);
   swapbyte(&ip.R);
   /* ----- inverse initial permutation ---- */
   inverse_permute(&op.L, &ip.L,
       (long *)IPtbl, 64);
   memcpy(blk, &op, sizeof(struct LR));
}

/* ---------- decrypt an 8-byte block ------------ */
void decrypt(char *blk)
{
   struct LR ip, op;
   long temp;
   int n;

   memcpy(&ip, blk, sizeof(struct LR));
   /* -------- initial permuation -------- */
   permute(&op.L, &ip.L, (long *)IPtbl, 64);
   swapbyte(&op.L);
   swapbyte(&op.R);
   ip.R = op.L;
   ip.L = op.R;
   /* ------ swap and key iterations ----- */
   for (n = 15; n >= 0; --n)    {
       temp = ip.L;
       ip.L = ip.R ^ f(ip.L, keys[n]);
       ip.R = temp;
   }
   swapbyte(&ip.L);
   swapbyte(&ip.R);
   /* ----- inverse initial permuation ---- */
   inverse_permute(&op.L, &ip.L,
       (long *)IPtbl, 64);
   memcpy(blk, &op, sizeof(struct LR));
}

/* ------- inverse permute a 64-bit string ------- */
static void inverse_permute(long *op,long *ip,long *tbl,int n)
{
    int i;
    long *pt = (long *)Pmask;

    *op = *(op+1) = 0;
    for (i = 0; i < n; i++)    {
       if ((*ip & *pt) ¦¦ (*(ip+1) & *(pt+1)))  {
           *op ¦= *tbl;
           *(op+1) ¦= *(tbl+1);
        }
        tbl += 2;
        pt += 2;
   }
}

/* ------- permute a 64-bit string ------- */
static void permute(long *op, long *ip, long *tbl, int n)
{
    int i;
    long *pt = (long *)Pmask;

    *op = *(op+1) = 0;
    for (i = 0; i < n; i++)    {
        if ((*ip & *tbl) ¦¦ (*(ip+1) & *(tbl+1))) {
            *op ¦= *pt;
            *(op+1) ¦= *(pt+1);
        }
        tbl += 2;
        pt += 2;
    }
}

/* ----- Key dependent computation function f(R,K) ----- */
static long f(long blk, struct ks key)
{
    struct LR ir;
    struct LR or;
    int i;
```

```
    union    {
        struct LR f;
        struct ks kn;
    } tr = {0,0}, kr = {0,0};

    ir.L = blk;
    ir.R = 0;

    kr.kn = key;

    swapbyte(&ir.L);
    swapbyte(&ir.R);

    permute(&tr.f.L, &ir.L, (long *)Etbl, 48);

    tr.f.L ^= kr.f.L;
    tr.f.R ^= kr.f.R;

   /*   the DES S function: ir.L = S(tr.kn);  */
    ir.L = 0;
    for (i = 0; i < 8; i++)    {
        long four = fourbits(tr.kn, i);
        ir.L |= four << ((7-i) * 4);
    }
    swapbyte(&ir.L);

    ir.R = or.R = 0;
    permute(&or.L, &ir.L, (long *)Ptbl, 32);

    swapbyte(&or.L);
    swapbyte(&or.R);

    return or.L;
}

/* ------- extract a 4-bit stream from the block/key ------- */
static int fourbits(struct ks k, int s)
{
    int i = sixbits(k, s);
    int row, col;
    row = ((i >> 4) & 2) | (i & 1);
    col = (i >> 1) & 0xf;
    return stbl[s][row][col];
}

/* ---- extract 6-bit stream fr pos s of the  block/key ---- */
static int sixbits(struct ks k, int s)
{
    int op = 0;
    int n = (s);
    int i;
    for (i = 0; i < 2; i++)    {
        int off = ex6[n][i][0];
        unsigned char c = k.ki[off];
        c >>= ex6[n][i][1];
        c <<= ex6[n][i][2];
        c &=  ex6[n][i][3];
        op |= c;
    }
    return op;
}

/* ---------- DES Key Schedule (KS) function ---------- */
static struct ks KS(int n, char *key)
{
    static unsigned char cd[8];
    static int its[] = {1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1};
    union    {
        struct ks kn;
        struct LR filler;
    } result;

    if (n == 0)
        permute((long *)cd, (long *) key, (long *)PC1tbl, 64);

    rotate(cd, its[n]);
    rotate(cd+4, its[n]);
    permute(&result.filler.L, (long *)cd, (long *)PC2tbl, 48);
    return result.kn;
}

/* rotate a 4-byte string n (1 or 2) positions to the left */
static void rotate(unsigned char *c, int n)
{
    int i;
    unsigned j, k;
    k = ((*c) & 255) >> (8 - n);
    for (i = 3; i >= 0; --i)    {
        j = ((*(c+i) << n) + k);
        k = (j >> 8) & 255;
        *(c+i) = j & 255;
    }
    if (n == 2)
       *(c+3) = (*(c+3) & 0xc0) | ((*(c+3) << 4) & 0x30);
    else
       *(c+3) = (*(c+3) & 0xe0) | ((*(c+3) << 4) & 0x10);
}

/* -------- swap bytes in a long integer ---------- */
static void swapbyte(long *l)
{
   char *cp = (char *) l;
   char t = *(cp+3);

   *(cp+3) = *cp;
   *cp = t;
   t = *(cp+2);
   *(cp+2) = *(cp+1);
   *(cp+1) = t;
}
```

End Listing Three

Listing Four

```
/* --------------- tables.c --------------- */
/* tables for the DES algorithm
 */

/* --------- macros to define a permutation table ---------- */
#define ps(n)       ((unsigned char)(0x80 >> (n-1)))
#define b(n,r)      ((n>r)!=(n<r-7)?0:ps(n-(r-8)))
#define p(n)        b(n, 8),b(n,16),b(n,24),b(n,32),\
                    b(n,40),b(n,48),b(n,56),b(n,64)
#define q(n)        p((n)+4)

/* --------- permutation masks ----------- */
unsigned char Pmask[] = {
    p( 1),p( 2),p( 3),p( 4),p( 5),p( 6),p( 7),p( 8),
    p( 9),p(10),p(11),p(12),p(13),p(14),p(15),p(16),
    p(17),p(18),p(19),p(20),p(21),p(22),p(23),p(24),
    p(25),p(26),p(27),p(28),p(29),p(30),p(31),p(32),
    p(33),p(34),p(35),p(36),p(37),p(38),p(39),p(40),
    p(41),p(42),p(43),p(44),p(45),p(46),p(47),p(48),
    p(49),p(50),p(51),p(52),p(53),p(54),p(55),p(56),
    p(57),p(58),p(59),p(60),p(61),p(62),p(63),p(64)
};

/* ----- initial and inverse-initial permutation table ----- */
unsigned char IPtbl[] = {
    p(58),p(50),p(42),p(34),p(26),p(18),p(10),p( 2),
    p(60),p(52),p(44),p(36),p(28),p(20),p(12),p( 4),
    p(62),p(54),p(46),p(38),p(30),p(22),p(14),p( 6),
    p(64),p(56),p(48),p(40),p(32),p(24),p(16),p( 8),
    p(57),p(49),p(41),p(33),p(25),p(17),p( 9),p( 1),
    p(59),p(51),p(43),p(35),p(27),p(19),p(11),p( 3),
    p(61),p(53),p(45),p(37),p(29),p(21),p(13),p( 5),
    p(63),p(55),p(47),p(39),p(31),p(23),p(15),p( 7)
};

/* ---------- permutation table E for f function --------- */
unsigned char Etbl[] = {
    p(32),p( 1),p( 2),p( 3),p( 4),p( 5),
    p( 4),p( 5),p( 6),p( 7),p( 8),p( 9),
    p( 8),p( 9),p(10),p(11),p(12),p(13),
    p(12),p(13),p(14),p(15),p(16),p(17),
    p(16),p(17),p(18),p(19),p(20),p(21),
    p(20),p(21),p(22),p(23),p(24),p(25),
    p(24),p(25),p(26),p(27),p(28),p(29),
    p(28),p(29),p(30),p(31),p(32),p( 1)
};

/* ---------- permutation table P for f function --------- */
unsigned char Ptbl[] = {
    p(16),p( 7),p(20),p(21),p(29),p(12),p(28),p(17),
    p( 1),p(15),p(23),p(26),p( 5),p(18),p(31),p(10),
    p( 2),p( 8),p(24),p(14),p(32),p(27),p( 3),p( 9),
    p(19),p(13),p(30),p( 6),p(22),p(11),p( 4),p(25)
};

/* --- table for converting six-bit to four-bit stream --- */
unsigned char stbl[8][4][16] = {
    /* ------------- s1 -------------- */
    14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7,
    0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8,
    4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0,
    15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13,
    /* ------------- s2 -------------- */
    15,1,8,14,6,11,3,4,9,7,2,13,12,0,5,10,
    3,13,4,7,15,2,8,14,12,0,1,10,6,9,11,5,
    0,14,7,11,10,4,13,1,5,8,12,6,9,3,2,15,
    13,8,10,1,3,15,4,2,11,6,7,12,0,5,14,9,
    /* ------------- s3 -------------- */
    10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8,
    13,7,0,9,3,4,6,10,2,8,5,14,12,11,15,1,
    13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7,
    1,10,13,0,6,9,8,7,4,15,14,3,11,5,2,12,
    /* ------------- s4 -------------- */
    7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15,
    13,8,11,5,6,15,0,3,4,7,2,12,1,10,14,9,
    10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4,
    3,15,0,6,10,1,13,8,9,4,5,11,12,7,2,14,
    /* ------------- s5 -------------- */
    2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9,
    14,11,2,12,4,7,13,1,5,0,15,10,3,9,8,6,
    4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14,
    11,8,12,7,1,14,2,13,6,15,0,9,10,4,5,3,
    /* ------------- s6 -------------- */
    12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11,
    10,15,4,2,7,12,9,5,6,1,13,14,0,11,3,8,
    9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6,
    4,3,2,12,9,5,15,10,11,14,1,7,6,0,8,13,
    /* ------------- s7 -------------- */
    4,11,2,14,15,0,8,13,3,12,9,7,5,10,6,1,
    13,0,11,7,4,9,1,10,14,3,5,12,2,15,8,6,
    1,4,11,13,12,3,7,14,10,15,6,8,0,5,9,2,
    6,11,13,8,1,4,10,7,9,5,0,15,14,2,3,12,
    /* ------------- s8 -------------- */
    13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7,
    1,15,13,8,10,3,7,4,12,5,6,11,0,14,9,2,
    7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8,
    2,1,14,7,4,10,8,13,15,12,9,0,3,5,6,11
};

/* ---- Permuted Choice 1 for Key Schedule calculation ---- */
unsigned char PC1tbl[] = {
    p(57),p(49),p(41),p(33),p(25),p(17),p( 9),
    p( 1),p(58),p(50),p(42),p(34),p(26),p(18),
    p(10),p( 2),p(59),p(51),p(43),p(35),p(27),
    p(19),p(11),p( 3),p(60),p(52),p(44),p(36),
    p(0),p(0),p(0),p(0),

    p(63),p(55),p(47),p(39),p(31),p(23),p(15),
    p( 7),p(62),p(54),p(46),p(38),p(30),p(22),
```

(continued on page 166)

Listing Four *(Listing continued, text begins on page 149.)*

```
    p(14),p( 6),p(61),p(53),p(45),p(37),p(29),
    p(21),p(13),p( 5),p(28),p(20),p(12),p( 4),
    p(0),p(0),p(0),p(0)
};

/* ---- Permuted Choice 2 for Key Schedule calculation ---- */
unsigned char PC2tbl[] = {
    p(14),p(17),p(11),p(24),p( 1),p( 5),p( 3),p(28),
    p(15),p( 6),p(21),p(10),p(23),p(19),p(12),p( 4),
    p(26),p( 8),p(16),p( 7),p(27),p(20),p(13),p( 2),

    q(41),q(52),q(31),q(37),q(47),q(55),q(30),q(40),
    q(51),q(45),q(33),q(48),q(44),q(49),q(39),q(56),
    q(34),q(53),q(46),q(42),q(50),q(36),q(29),q(32)
};

/* ---- For extracting 6-bit strings from 64-bit string ---- */
unsigned char ex6[8][2][4] = {
    /* byte, >>, <<, & */
    /* ---- s = 8  ---- */
    0,2,0,0x3f,
    0,2,0,0x3f,
    /* ---- s = 7  ---- */
    0,0,4,0x30,
    1,4,0,0x0f,
    /* ---- s = 6  ---- */
    1,0,2,0x3c,
    2,6,0,0x03,
    /* ---- s = 5  ---- */
    2,0,0,0x3f,
    2,0,0,0x3f,
    /* ---- s = 4 ---- */
    3,2,0,0x3f,
    3,2,0,0x3f,
    /* ---- s = 3 ---- */
    3,0,4,0x30,
    4,4,0,0x0f,
    /* ---- s = 2 ---- */
    4,0,2,0x3c,
    5,6,0,0x03,
    /* ---- s = 1 ---- */
    5,0,0,0x3f,
    5,0,0,0x3f
};
```

End Listing Four

Listing Five

```
/* ---------------------- encrypto.c ------------------------ */
/* Single key text file encryption
 * Usage: encrypto keyvalue infile outfile
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define FALSE 0
#define TRUE !FALSE

static void charout(FILE *fo, char prev, int runct, int last);
static void encrypt(FILE *fo, char ch, int last);

static char *key = NULL;
static int keylen;
static char *cipher = NULL;
static int clen = 0;

void main(int argc, char *argv[])
{
   FILE *fi, *fo;
   char ch, prev = 0;
   int runct = 0;

   if (argc > 3)    {
       /* --- alloc memory for the key and cipher blocks --- */
       keylen = strlen(argv[1]);
       cipher = malloc(keylen+1);
       key = malloc(keylen+1);
       strcpy(key, argv[1]);

       if (cipher != NULL && key != NULL &&
               (fi = fopen(argv[2], "rb")) != NULL)        {
           if ((fo = fopen(argv[3], "wb")) != NULL)    {
               while ((ch = fgetc(fi)) != EOF)    {
                   /* ---- validate ASCII input ---- */
                   if (ch & 128)    {
                       fprintf(stderr, "%s is not ASCII",
                                   argv[2]);
                       fclose(fi);
                       fclose(fo);
                       remove(argv[3]);
                       free(cipher);
                       free(key);
                       exit(1);
                   }

                   /* --- test for duplicate bytes --- */
                   if (ch == prev && runct < 127)
                       runct++;
                   else    {
                       charout(fo, prev, runct, FALSE);
                       prev = ch;
                       runct = 0;
                   }
               }
               charout(fo, prev, runct, TRUE);
               fclose(fo);
           }
           fclose(fi);
       }
       if (cipher)
           free(cipher);
       if (key)
           free(key);
   }
}

/* ------- send an encrypted byte to the output file ------ */
static void charout(FILE *fo, char prev, int runct, int last)
{
   if (runct)
       encrypt(fo, (runct+1) | 0x80, last);
   if (prev)
       encrypt(fo, prev, last);
}

/* ---------- encrypt a byte and write it ---------- */
static void encrypt(FILE *fo, char ch, int last)
{
   *(cipher+clen) = ch ^ *(key+clen);
   clen++;
   if (last || clen == keylen)    {
       /* ----- cipher buffer full or last buffer ----- */
       int i;
       for (i = 0; i < clen; i++)
           fputc(*(cipher+i), fo);
       clen = 0;
   }
}
```

End Listing Five

Listing Six

```
/* ---------------------- decrypto.c ------------------------ */
/* Single key text file decryption
 * Usage: decrypto keyvalue infile outfile
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <process.h>
static char decrypt(FILE *);

static char *key = NULL;
static int keylen;
static char *cipher = NULL;
static int clen = 0, coff = 0;

void main(int argc, char *argv[])
{
   FILE *fi, *fo;
   char ch;
   int runct = 0;

   if (argc > 3)    {
       /* --- alloc memory for the key and cipher blocks --- */
       keylen = strlen(argv[1]);
       cipher = malloc(keylen+1);
       key = malloc(keylen+1);
       strcpy(key, argv[1]);

       if (cipher != NULL && key != NULL &&
               (fi = fopen(argv[2], "rb")) != NULL)    {

           if ((fo = fopen(argv[3], "wb")) != NULL)    {
               while ((ch = decrypt(fi)) != EOF)    {
                   /* --- test for run length counter --- */
                   if (ch & 0x80)
                       runct = ch & 0x7f;
                   else    {
                       if (runct)
                           /* --- run count: dup the byte -- */
                           while (--runct)
                               fputc(ch, fo);
                       fputc(ch, fo);
                   }
               }
               fclose(fo);
           }
           fclose(fi);
       }
       if (cipher)
           free(cipher);
       if (key)
           free(key);
   }
}

/* ------ decryption function: returns decrypted byte ----- */
static char decrypt(FILE *fi)
{
   char ch = EOF;
   if (clen == 0)    {
       /* ---- read a block of encrypted bytes ----- */
       clen = fread(cipher, 1, keylen, fi);
       coff = 0;
   }
   if (clen > 0)    {
       /* --- decrypt the next byte in the input block --- */
       ch = *(cipher+coff) ^ *(key+coff);
       coff++;
       --clen;
   }
   return ch;
}
```

End Listings

Listing One *(Text begins on page 155.)*

```
FUNCTION Modulus(X,Y : Integer) : Integer;

VAR
  R : Real;

BEGIN
  R := X/Y;
  IF R < 0 THEN
    Modulus := X-(Y*Trunc(R-1))
  ELSE
    Modulus := X-(Y*Trunc(R));
END;
```

End Listing One

Listing Two

```
PROGRAM ZelTest2;  { From DDJ 11/90 }

CONST
  DayStrings : ARRAY[0..6] OF STRING =
  ('Sunday','Monday','Tuesday','Wednesday','Thursday','Friday','Saturday');

VAR
  Month, Day, Year : Integer;

{ This function implements true modulus, rather than }
{ the remainder function as implemented in MOD.      }

FUNCTION Modulus(X,Y : Integer) : Integer;

VAR
  R : Real;

BEGIN
  R := X/Y;
  IF R < 0 THEN
    Modulus := X-(Y*Trunc(R-1))
  ELSE
    Modulus := X-(Y*Trunc(R));
END;

FUNCTION CalcDayOfWeek(Year,Month,Day : Integer) : Integer;

VAR
  Century,Holder : Integer;

BEGIN
  { First test for error conditions on input values: }
  IF (Year < 0)  OR
     (Month < 1) OR (Month > 12) OR
     (Day < 1)   OR (Day > 31) THEN
     CalcDayOfWeek := -1  { Return -1 to indicate an error }
  ELSE
    { Do the Zeller's Congruence calculation as Zeller himself }
    { described it in "Acta Mathematica" #7, Stockhold, 1887.  }
    BEGIN
      { First we separate out the year and the century figures: }
      Century := Year DIV 100;
      Year    := Year MOD 100;
      { Next we adjust the month such that March remains month #3, }
      {  but that January and February are months #13 and #14,     }
      {  *but of the previous year*: }
      IF Month < 3 THEN
        BEGIN
          Inc(Month,12);
          IF Year > 0 THEN Dec(Year,1)      { The year before 2000 is }
            ELSE                            { 1999, not 20-1...       }
              BEGIN
                Year := 99;
                Dec(Century);
              END
        END;

      { Here's Zeller's seminal black magic: }
      Holder := Day;                        { Start with the day of month }
      Holder := Holder + (((Month+1) * 26) DIV 10); { Calc the increment }
      Holder := Holder + Year;              { Add in the year }
      Holder := Holder + (Year DIV 4);      { Correct for leap years  }
      Holder := Holder + (Century DIV 4);   { Correct for century years }
      Holder := Holder - Century - Century; { DON'T KNOW WHY HE DID THIS! }

      Holder := Modulus(Holder,7);          { Take Holder modulus 7  }

      { Here we "wrap" Saturday around to be the last day: }
      IF Holder  = 0 THEN Holder := 7;

      { Zeller kept the Sunday = 1 origin; computer weenies prefer to }
      { start everything with 0, so here's a 20th century kludge:     }
      Dec(Holder);

      CalcDayOfWeek := Holder;  { Return the end product! }
    END;
END;

BEGIN
  Write('Month (1-12): '); Readln(Month);
  Write('Day   (1-31): '); Readln(Day);
  Write('Year        : '); Readln(Year);
  Writeln('The day of the week is ',
          DayStrings[CalcDayOfWeek(Year,Month,Day)]);
  Readln;
END.
```

End Listings

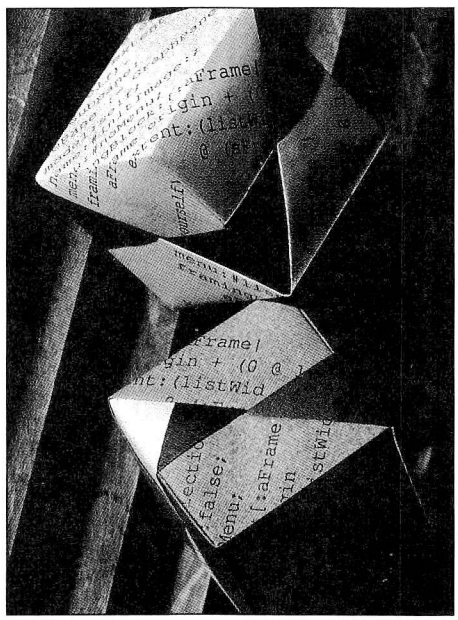
The MVC Paradigm in Smalltalk/V

Model-View-Controller becomes Object-Pane Dispatcher

Kenneth E. Ayers

My recent efforts at unraveling more of Smalltalk's mysteries centered around the windowing system. After creating a few applications and spending many, many hours browsing through the source code (thank heaven for the source code!), I finally began to form a mental picture of how the windowing system behaves. The picture is not a simple one. For instance, the example code presented in this article was extracted from the six pages of code involved just in opening a window.

I'll begin by describing the Model-View-Controller (MVC) paradigm as implemented in classic Smalltalk. Secondly, I'll describe how the architecture under Smalltalk/V286 differs from the classic paradigm. Finally, I'll focus in more detail on the mechanisms of creating, opening, and framing a window.



The MVC Paradigm

The classic Smalltalk system, as designed at Xerox PARC in the 1970s, is based on the Model-View-Controller paradigm, the conceptual framework of which was discussed in "Information Models, Views, and Controllers," by Adele Goldberg (*DDJ*, July 1990).

An application in Smalltalk-80 consists of three components: A model that produces the information, a view that displays it, and a controller that manages input events.

The model is in some sense the core of the application, the data structures that represent what the application is trying to accomplish. A view is an object that actually presents the information contained in a model on the display screen. Various types of view objects are provided by the system, each designed to display specific types of information. For example, one kind of view can display data as a scrollable list, while another presents text so it can be edited.

Controllers exist to process input from the keyboard and mouse. Generally speaking, for each type of view there is a corresponding type of controller. For example, a list controller recognizes particular input events as indicating a selection from the displayed list of items. The text editor's controller handles specific events for marking, cutting, or pasting a block of text or for scrolling a page.

In the Smalltalk-80 implementation, the MVC architecture is embodied in the hierarchies for the classes *Model, View,* and *Controller.* The implementors of Digitalk's Smalltalk/V286 concentrated the functionality of the MVC paradigm into a smaller set of more complex classes. The mechanisms are different enough that the two platforms are largely incompatible at the application level (even though the language implementations are almost identical at the syntactic level).

How Smalltalk/V and Smalltalk-80 Differ

The most obvious difference between Smalltalk/V and Smalltalk-80 is in the class names. First of all, class *Model* is gone! This isn't a serious problem. Ultimately, the application *is* the model; and the dependent notification mechanisms are available anyway — through class *Object.*

In Smalltalk/V286, a view object is an instance of class *Pane* or one of its subclasses. Figure 1 shows the basic

Ken is a software engineer at Eaton/IDT in Westerville, Ohio. He is involved in the design of real-time software for industrial graphic workstations. He also works part time as a consultant, specializing in prototyping custom-software systems and applications. Ken can be contacted at 7825 Larchwood St., Dublin, OH 43017.

(continued from page 168)
hierarchy for class *Pane* and Table 1 describes its principal classes. Similarly, controllers are derived from class *Dispatcher*, which is detailed in Figure 2 and Table 2.

Smalltalk/V286 has several classes and global objects that provide high-level management functions for the window system. One of these is the global variable *Display* — the only instance of class *DisplayScreen*. *Display* represents the physical display and provides access to the screen as a bitmap.

Another major player is the global object *Scheduler*. The only allowable instance of class *DispatchManager*, *Scheduler* maintains a list of all the windows on the screen and manages their activation and deactivation.

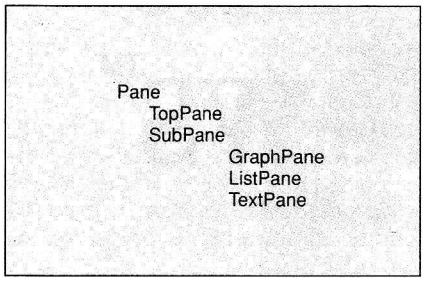


Figure 1: *The hierarchy for* Pane *provides protocols for displaying information within a window.*

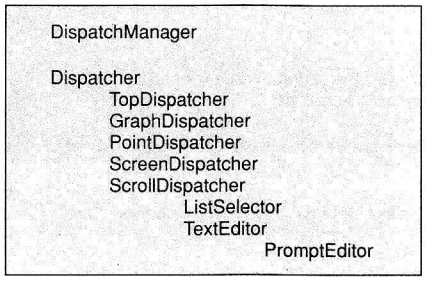


Figure 2: *The hierarchy for* Dispatcher *provides protocols for handling input events directed to a window.*

Pane	Provides the common instance variables and the default behaviors for its subclasses, such as drawing borders and popping up menus.
TopPane	Represents the window itself, embodied by the border and the title bar.
SubPane	Represents independent regions within a window. *SubPane* provides the common behaviors such as reframing for its subclasses.
TextPane	Provides methods for scrolling text and notifying the model when its contents change or are saved.
ListPane	Supports scrolling and displaying data in the form of a list of individual items; provides methods to visually indicate the current selection.
GraphPane	A generalized "canvas" for graphic drawing.

Table 1: *Description of the principal* Pane *classes*

The subpane's dispatcher enters its own control loop to monitor input events

The global variable *Processor* is the only instance of class *ProcessScheduler*, whose responsibility it is to manage the various processes that might be active in the system at any given time. Among these is the user interface process that fields interrupts from the keyboard and the mouse and translates them into suitable input events.

Raw input is handed off to the global object *Terminal* — an instance of class *TerminalStream*. *Terminal* takes care of the housekeeping associated with the input stream. Internally, its state machine maps multistage input events (mouse movements, button clicks, key scan codes, and so on) into a set of global event function codes (which are defined in the pool dictionary *FunctionKeys*).

Finally, the global variable *Transcript* is an instance of class *TextEditor*. *Transcript* is used much like a console window to relay error or status information. However, being a generic text editor, *Transcript* is also available for use in evaluating any Smalltalk expression, via the "do it" or "show it" menu selections.

The Chain of Command

Each of the application's panes, including the top pane, has a *Dispatcher* associated with it for handling input events. Input comes into the application via the active dispatcher. How is this dispatcher activated?

First, let's assume that the cursor is initially positioned on the background screen, outside any window. Under these conditions, the system scheduler (*Scheduler*) is calling the shots. Basically, *Scheduler* sits in a loop and steps through its list of windows, sending

(continued from page 170)
each one the message *isControlWanted* (which, by default, merely tests to see if the cursor is positioned within the window's borders).

The first window that answers "yes" becomes the active window. *Scheduler* puts that window at the head of its list and sends its dispatcher (usually a *TopDispatcher*) the message *activateWindow*.

DispatchManager	Manages all of the windows on the screen. There is only one instance of this class: the global variable *Scheduler*.
Dispatcher	Provides the default behaviors for processing input from the mouse or keyboard.
TopDispatcher	Provides methods to process inputs, including cursor positioning and menu activation, directed at the window itself.
ScrollDispatcher	Provides the default behavior for processing input related to scrolling the image in a pane.
TextEditor	Processes input for its associated *TextPane*.
ListSelector	Processes input for its associated *ListPane*.
GraphDispatcher	Processes input directed to a *GraphPane*.

Table 2: *Description of the principal* Dispatcher *classes*

reframePane:aPane	Sent by a *GraphPane* when the size or position of a window is changed. The model should implement this message if there is more than one *GraphPane* in a window, because it provides the identity of the pane being reframed. The actual frame can be determined by the message *aPane frame*.
reframe:aRectangle	Also sent by a *GraphPane* when the window is reframed. The model should implement this message when there is only one *GraphPane* in the window.
showWindow	Sent by the *TopPane* when the window's contents must be refreshed for any reason. Serves as an indication that the window is being displayed and that any application information should be prepared for presentation.
activatePane	Sent by a *GraphPane* when the cursor enters its frame. This message can be used, for example, to change the cursor's shape so that the user receives a visual indication that the cursor has entered a particular region.
deactivatePane	Sent by a *GraphPane* when the cursor leaves its frame.
close	Sent when the window is being closed. This is useful for saving any unfinished business before the window disappears!
label	Sent when the model is expected to answer the string that will be used for the window's label.
collapsedLabel	Sent when the model should answer the string that it wants used to label the collapsed window. Receipt of this message can also indicate that the window is in the process of being collapsed.
InitWindowSize	Sent by the *open* method in class *Dispatcher*. This message gives the model an opportunity to specify the initial size of the window.

Table 3: *Methods optionally implemented by Model-to-Handle user-generated events*

closeIt	Closes the window.
zoom	Normally applies only to instances of class *TextPane*, which will then expand to the full screen-size. If window contains no text panes, the *zoom* message will be sent to the model.
reframe	Indicates that the user wishes to change the size of the window.
collapse	Causes the window to collapse to an iconic form (an abbreviated title bar containing only the window's label).

Table 4: *Methods optionally implemented by Model-to-Handle icon button events*

Once a *TopDispatcher* acquires control, it enters its own control loop. Here, the *TopDispatcher* polls the dispatchers associated with each of the window's subpanes, asking if one of them wants control. The pane that contains the cursor is then marked as the active pane and its dispatcher assumes control of the input stream.

When the dispatcher for a subpane gains control, it goes through an activation sequence which, in the case of a *GraphPane*, for instance, includes sending the message *activatePane* to its model.

Finally, the subpane's dispatcher enters its own control loop to monitor input events. The specific type of dispatcher that ultimately gets control determines the nature of the system's reaction to input events.

Keeping Everybody Informed

Much of the power (and the complexity) of the Smalltalk/V286 windowing system is a result of the dialogue between a window (the pane and/or dispatcher) and its model. For almost every type of event that a user can generate, there is a mechanism for notifying the model about potential changes. Some events are handled transparently by the pane or its dispatcher, so they don't "bother" the application. However, from the viewpoint of a model's implementation, these event response capabilities are considered optional. Consequently, in most cases, the pane or dispatcher "asks" the model about its implementation, using a sequence of statements similar to that in Example 1(a).

The methods that a model may implement to handle window related events are listed in Table 3.

In addition to the messages described in Table 3, still other messages may be sent to the model by selecting one of the iconic "buttons" that appear in a window's title bar. The actions associated with these buttons, such as closing, collapsing, moving, or resizing, are directly related to the behavior of window. However, these same actions may also affect the way the model carries out its internal operations.

Therefore, when the top dispatcher detects a left mouse button click with the cursor positioned inside one of these buttons, it fetches the name of the icon (a *Symbol*) from a class variable. The icon's name is actually the name of a method that is supposed to perform the action. If the model is capable of responding to the message, it will be sent; otherwise, the top dispatcher performs some default operation. These messages are described in Table 4.

All of the messages mentioned in

Tables 3 and 4 are sent to the model by the window — typically as the result of a user-initiated event. There are also many ways in which the model can influence the configuration and operation of its windows. In fact, much of the public protocol for the *Dispatcher* and *Pane* class hierarchies is available for just this purpose. Some of the more important messages involved in these operations are given in Table 5.

Constructing a Window

The first step in constructing an application window at runtime is to create the top pane for the window. This step, illustrated by the open method found in Listing One (page 175), is marked by the line: *topPane := TopPane new.* Following this is a series of cascaded messages, directed at the newly created *topPane*, that specify the initial setup for the window (its associated model, label, menu, and so on).

Once the *topPane* has been taken care of, the subpanes can be created and added to the *topPane* as in Example 1(b). Here the cascaded messages provide setup information to the subpane.

Notice that in Listing One there are three subpanes created and added to the *topPane*. One of these is a *ListPane*, while the other two are *GraphphPanes*. Also notice that each subpane is assigned a "name" by the statement: *name:< aSymbol >;*

This can be confusing, because the name given is not really the "name" of the subpane. It is, in fact, the name of a method selector that will be sent to the model during the process of opening the window. This method is supposed to perform application-specific initialization for the pane. In the case of the *ListPane*, the method should answer an array containing the list of items to be displayed. For a *GraphPane*, the initialization method is expected to answer a Form (bitmap) whose size is the same as the pane's frame (the frame is passed as an argument to the message).

Framing a Window

Framing a window is one of the more obscure parts of creating a window application in Smalltalk/V286. Each subpane must possess a means of determining its position and size relative to

```
(a) (model respondsTo:#activatePane)
      ifTrue:[model perform:#activatePane].

(b) topPane addSubPane:
     (aPane := GraphPane new
       <cascaded messages>;
       yourself).
```

Example 1: *Creating and adding subpanes*

label:aString	Sent to a *TopPane* to specify the label which should appear at the top of the window.
model:anObject	Notifies a *Pane* that its controlling model is *anObject*.
name:aSymbol	Tells a *Pane* that *aSymbol* is the name of the method, implemented in the model class, that provides initialization for the pane. The specified method may or may not be expected to accept an argument, depending upon the type of pane. For example, a rectangle (the pane's frame) will be passed as an argument to the "name" method associated with a *GraphPane*.
menu:aSymbol	Supplies a *Pane* with the name of a method that will answer a *Menu* substituted for the pane's default menu.
change:aSymbol	Here, *aSymbol* is the name of a single-argument method that handles user-initiated selections (such as pressing the left mouse button) within the pane.
selectUp:aSymbol	For *GraphPane* only, *aSymbol* is the name of a message that is sent when the left mouse button is released.
framingRatio:aRectangle	Used to specify the region of the window's display that the subpane will occupy.
framingBlock:aBlock	Used to specify, more precisely, the position and size of a subpane within the window.

Table 5: *Messages optionally sent by the Model to configure a window*

the window as a whole. The calculations must be such, that if the window's position or size is allowed to change, the position and size of each subpane can be adjusted to accommodate the new frame.

There are two ways of framing subpanes. Either of the messages *framingRatio:aRectangle* or *framingBlock: aBlock* can be used, depending upon the level of precision required. If the absolute position or size of the subpanes is not critical, *framingRatio:* is far easier to use. This message specifies the position and size of a subpane as fractions of the whole window. For example, *framingRatio:(0 @ (3/4) extent:1/4 @ (1/4));* tells a subpane that it will be positioned at the left edge of the window and three quarters of the way down from the top. The subpane's size will be one quarter of the width of the window and one quarter of its height. In other words, it will occupy the lower left-hand corner of the parent window.

However, in those cases when an application demands that one or more of its subpanes be located in a specific position or be framed to a certain absolute size, the procedure becomes a bit more complicated. Listing Two (page 175) is a modified version of the open method in which the list pane is constrained to be ten characters wide and ten lines high. Here the message *framingBlock:* is sent to each subpane with a block of code as its argument. The block of code is *not* evaluated at the time this message is sent to the subpane, but when the window is opened or reframed. At that time, it is passed a single argument — a rectangle defining the the window's interior frame. The block should answer a new rectangle — the frame of the subpane.

Any variables local to the open method that are referenced within the framing block, will have the values they had when the open method completed. Be careful with this one — don't try to calculate the dimensions of the subpane frames incrementally, using one set of variables. The results will not be what you expect!

Wrapping Up

There's no doubt that Smalltalk is a big and complex system and that the powerful array of features it offers can be quite intimidating. Fortunately, the environment provides the kinds of tools that make exploration considerably less difficult.

DDJ

(Listings begin on page 175.)

Vote for your favorite feature/article.
Circle Reader Service **No. 16.**

Listing One *(Text begins on page 168.)*

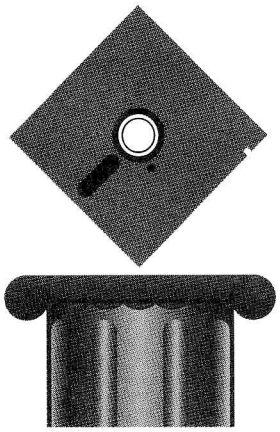
```
open
    | frame |

    appName   := String new.
    saved     := true.
    editorPen := Pen new.
    imagePen  := Pen new.
    frame     := (Display boundingBox extent // 6)
                    extent:(Display boundingBox extent * 2 // 3).
    topPane := TopPane new
        model:self;
        label:self label;
        menu:#windowMenu;
        minimumSize:frame extent;
        yourself.
    topPane addSubpane:
        (listPane := ListPane new
            model:self;
            name:#appList;
            change:#appSelection:;
            returnIndex:false;
            menu:#listMenu;
            framingRatio:(0 @ 0 extent:1/4 @ (2/3));
            yourself).
    topPane addSubpane:
        (imagePane := GraphPane new
            model:self;
            name:#initImage:;
            menu:#noMenu;
            framingRatio:(0 @ (2/3) extent:1/4 @ (1/3));
            yourself).
    topPane addSubpane:
        (editorPane := GraphPane new
            model:self;
            name:#initEditor:;
            menu:#editorMenu;
            change:#editIcon:;
            framingRatio:(1/4 @ 0 extent:3/4 @ 1);
            yourself).
    topPane reframe:frame.
    topPane dispatcher openWindow scheduleWindow.
```

End Listing One

Listing Two

```
    open
        | frame listWid listHgt |

        saved     := true.
        editorPen := Pen new.
        imagePen  := Pen new.
        frame     := (Display boundingBox extent // 6)
                        extent:(Display boundingBox extent * 2 //
    3).
        listWid   := SysFont width * 10.
        listHgt   := SysFont height * 10.


        topPane := TopPane new
            label:self label;
            model:self;
            menu:#windowMenu;
            minimumSize:frame extent;
            yourself.
        topPane addSubpane:
            (listPane := ListPane new
                model:self;
                name:#appList;
                change:#appSelection:;
                returnIndex:false;
                menu:#listMenu;
                framingBlock:[:aFrame |
                    aFrame origin
                        extent:listWid @ listHgt];
                yourself).
        topPane addSubpane:
            (imagePane := GraphPane new
                model:self;
                name:#initImage:;
                menu:#noMenu;
                framingBlock:[:aFrame|
                    aFrame origin + (0 @ listHgt)
    extent:(listWid
                                @ (aFrame height - listHgt))];
                yourself).
        topPane addSubpane:
            (editorPane := GraphPane new
                model:self;
                name:#initEditor:;
                menu:#editorMenu;
                change:#editIcon:;
                framingBlock:[:aFrame|
                    aFrame origin + (listWid @ 0)
                        extent:((aFrame width - listWid)
                                    @ aFrame height)];
                yourself).
        topPane reframe:frame.

        topPane dispatcher openWindow scheduleWindow.
```

End Listings



Phase One is a C engine that replaces Borland's Paradox C Engine. **TSR Systems** claims that Phase One is four to ten times faster than any other currently available Paradox C engine. Enhanced memory management routines and file management algorithms are the basis for Phase One.

DDJ spoke with a TSR consultant, Dennis Young, who's using the engine to develop a report writer. He described the engine as "a way station — these functions are things we needed in order to compile PAL scripts. The engine kind of fell out of the development of the compiler, and wound up being much faster than the Paradox Engine."

The DOS version requires a Microsoft C-compatible compiler or Borland's Turbo C or Turbo C++. The DLL for Windows 3.0 should now be available, also. The engine contains both the DOS and Windows libraries and retails for $495, though any owner of a competitive Paradox C engine can upgrade for $149. Reader service no. 20.
TSR Systems Limited
116 Oakland Ave.
Port Jefferson, NY 11777
516-331-6336

The Spinnaker PLUS Software Slot Developer's Kit for the Macintosh has been released by **Spinnaker Software**. With it you can create customized extensions to the PLUS development environment. With Spinnaker PLUS you can develop applications that look and run the same, without modification, across Windows 3.0, OS/2 Presentation Manager, and Macintosh operating systems.

With Software Slots, PLUS can accept new object classes and scripting language extensions. Commands and functions similar to external commands and functions extend the PLUS programming language and allow an unlimited number of arguments and customized syntax. External draw objects add new object types such as a clock, a digital instrument display, a bar graph, a 3-D chart, and other data representations. The Kit is available for $695. Reader service no. 26.
Spinnaker Software
201 Broadway
Cambridge, MA 02139-1901
617-494-1200

A development tool for Windows comes from **EdenSoft**. Resource Workshop includes such resources as menus, icons, fonts, cursors, bitmaps, string tables, accelerators, and dialogs.

DDJ spoke with David Rowley of Varian Associates, developers of Windows-compatible products, who's been involved in the beta test since the beginning. Rowley claims "It's better than any tool Microsoft provides for their software developers; it allows quick prototyping of resources for an application — you can use it as a development tool without having to edit the resource script file. You can even edit from within EXE files."

Other features are a Project View with all related files and resources, incremental compilation, full source code preservation, language extensions, multiple undo and redo commands, the ability to edit resources as text or graphically, extensive support of #defines, automatic usage cross reference, and support for both Windows 2 and 3 resources. And Resource Workshop is compatible with older tools. It sells for $295. Reader service no. 22.
EdenSoft
2980 College, Suite 7
Berkeley, CA 95705
415-548-3554

SATVU, a satellite propagation and field-of-view simulation software package for PCs, is new from **Applied Research**. SATVU animates the field-of-view seen from a satellite, ground station, rocket trajectory, or point in inertial space. From any of these points you can see other satellites, the Earth, sun, moon, planets, and more than 1700 stars and deep-sky objects.

DDJ spoke with Harold Fears, the senior engineer on the project, who told us he wrote the time-critical routines in assembler, the menus in Turbo C, and debugged with the Turbo Debugger. "We adapted this program from a larger one we developed for the Defense Department, and are targeting a smaller market, such as ham operators, amateur astronomers, and anyone interested in satellites, planets, and stars. It's all menu-driven, though you can use a mouse, and it only takes about 15 minutes to learn."

You need an AT or 386 compatible with math coprocessor, hard disk, and EGA or VGA graphics. The animation update rate is approximately one screen per second on a 20-MHz 386. Applied Research can customize a version for you — they have different subroutines and modules that they can add to the product. As is, SATVU sells for $200. Reader service no. 25.
Applied Research Inc.
5025 Bradford Blvd.
Cummings Research Park
Huntsville, AL 35805
205-837-8600

Objectworks\C++ Release 2 from **ParcPlace Systems** is now available for the Sun-3 and SPARCstation platforms. An integrated development system designed to help C++ programmers take better advantage of object-oriented programming by creating and reusing C++ code. New features let you work with Objectworks\C++ in conjunction with traditional Unix tools. The product supports large system development and incorporates AT&T C++ Language System, Release 2.1. C++ class libraries allow for code reuse.

The open environment of Objectworks\C++ lets you use your favorite C preprocessor, C compiler, and linker to customize your program. It is also compatible with third party source code control systems, profilers, and debuggers. The source-level process inspector allows debugging of C as well as C++ code. Integration into a single, window-based environment means you no longer have to switch between editors, grep, compilers, linkers, and debuggers, making code development faster. Objectworks\C++ costs $3000, and ObjectKit\C++, a collection of reusable class libraries, costs $500. Reader service no. 24.
ParcPlace Systems
1550 Plymouth St.
Mountain View, CA 94043
415-691-6700

ED the Programmer's Editor, from the Australian company **Soft As It Gets**, was designed to simplify and speed up program writing and development. A virtual memory, text-editing engine with a full C extension language allows you to extend the editor's capabilities. A C interpreter and companion compiler are built in, and ED supports both C and C++. ED includes such features as fast look-up of functions and methods, smart indenting and templates, direct access to include files, a C function browser, and full undo/redo.

ED is fully configurable, including

(continued on page 182)

(continued from page 177)
keys for initiating commands, window colors, and user-defined menus. ED was designed to support programming teams, and includes LAN multiuser locking, which allows one user to edit a file while others view it. Multiprogrammer support allows each user to have their own menus, configuration information, and keyboard bindings. ED fits into 128K of memory, and can swap to disk or EMS if necessary. Contact the company for pricing. Reader service no. 23.
Soft As It Gets
3 Pullman Ct.
East St. Kilda
3183 Victoria, Australia

Cedar Software has released Fractal Grafics, for creating complex images with fractal geometry and a visual programming language. You design a template and the program continues your pattern automatically. If, for example, you draw the trunk and first few branches of a tree, the program will draw the rest. You can load and save images in PCX format or as highly compressed fractal templates.

With either a mouse or a keyboard you can spin, skew, grow, shrink, stretch, squish, and rearrange parts of any shape without losing texture or detail. Changes can then be reflected through all levels. The program features an online interactive tutorial, point and click menus, and full color control. The guidebook extensively explains and illustrates the background of fractals as well as all formulae and algorithms. Full source code is available, and the package sells for $79. Reader service no. 31.
Cedar Software
RR 1, Box 5140
Morrisville, VT 05661-5140
802-888-5275

A comprehensive user interface development package for Smalltalk/V 286 is available from **Acumen Software**. Widgets/V 286 is an interactive editor for point and click development, and an extensive user interface object library for building block reusability. To build an interface, you move, size, and edit the widgets directly on the screen. You can switch back and forth between designing and testing with the editor's run mode, for rapid prototyping.

DDJ spoke with Eric Langjahr, president of Impeccable Software and beta tester for Widgets, who said, "Widgets provides a very large class library for building user interfaces that are more modern than any available for PCs — similar to the NextStep, with three-dimensional images and the like. You build visually and put the programmatic pieces in later. It's unique to anything available now."

The object library includes a broad spectrum of common visual objects for use in graphical user interfaces, such as various window styles, hierarchical pop-up and pull-down menus, list boxes, text editor, scrolling pages, radio buttons, and palettes of forms. A set of intrinsics is provided for creating customized widgets. Widgets/V is selling for $99. Reader service no. 29.
Acumen Software
851 Lytton
Palo Alto, CA 94301
415-328-3816

Another product available for Smalltalk users is MathPac1/V from **Knowledge Systems**. MathPac1/V lets you perform advanced mathematical operations within Smalltalk applications. The ability to work within the same development environment instead of using another language should reduce development time.

MathPac1/V features vectors, two-dimensional matrices, N-dimensional matrices, complex numbers, long-cycle random number generation, beta function, incomplete beta function, in of gama function, binomial coefficient, and string asFloat. MathPac1/V is priced at $199. Reader service no. 30.
Knowledge Systems Corp.
114 MacKenan Dr., Ste. 100
Cary, NC 27511-6446
919-481-4000

GEOGRAF Level One from **GEOCOMP** is a graphics library of subroutines and functions for creating custom graphs and charts. For use with most Microsoft and Borland compilers, GEOGRAF Level One enables you to add graphics to programs without having to develop complex graphics device drivers for each device you wish to support, and you can output your graphics to any output device. Because GEOGRAF Level One uses the same library structure as most of the Microsoft and Borland compilers, you can use similar graphics calls with different languages and compilers. Device drivers can be changed at any time without changing the software.

DDJ spoke with Doug McGary, project engineer at TMA Technologies and beta tester for GEOGRAF Level One. "I'm involved in image processing [and] needed to be able to detect defects in a special plastic material used for cornea replacement. I figured out the subroutine calls I needed to make and wrote the code and had it debugged in three days. GEOGRAF saved me three man-weeks of software writing — it's well documented and easy to use."

GEOGRAF Level One sells for $149 and comes with a 30-day money-back guarantee. Reader service no. 27.
GEOCOMP Corporation
66 Commonwealth Ave.
Concord, MA 01742
800-822-2669

A conference examining the latest in technological tools for creating visuals, sound, live performance, and integrated media experiences took place at **CyberArts International** in Los Angeles in September. CyberArts was a showcase of artists, programmers, musicians, special effects technicians, and multimedia gurus combining technologies in film, music, and art production. The three-day affair consisted of concerts and performances, workshops, presentations, exhibits, and an art gallery.

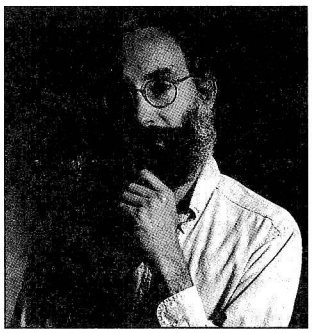
The list of speakers was a who's who in this eclectic, interconnected, emerging field. Jeff Rona and Chris Meyer demonstrated "The Music Cognition Link," a HyperCard application built on inference engines created through Lisp. Based on a spinoff of neural nets with back propagation, this program accepts input from a keyboard and responds in kind. The level of the computer's response was surprisingly avant-garde — at times it seemed as if the computer was more creative than its human counterparts.

In other presentations Jaron Lanier discussed virtual reality, Marc Canter answered the question "What the Heck Is Multimedia?", Bill Buxton blasted the distinction between artists and technologists, Allen Adkins discussed the impact of the CD David Zicarelli demonstrated the basics of interactive creativity, Carl Rodendahl presented "live" animation (digital puppet techniques), and Kit Galloway and Sherrie Rabinowitz of the Electronic Cafe talked about "composite-image performances," which they make possible through a satellite link. Other presentations demonstrated interactive toys, hyperinstruments, and desktop computer animation.

Next year's conference, also to be sponsored by *Keyboard* magazine and Miller Freeman, is scheduled for August 22-25, again in Los Angeles.

DDJ

How To Succeed In the Window Business Without Really Trying



Software author and Windows developer Alan Cooper recently asked me to appear on a Software Entrepreneur's Forum panel whose mission was to discuss making money with Windows 3.0. It proved impossible for me to attend, but I prepared some remarks anyway. For what they're worth, here are my thoughts on making money with Windows 3.0. You are welcome to disagree with them, as a whole or individually.

Windows 3.0 is being marketed as a new platform. Given that, all the new-platform truisms apply; in particular, the get-there-first truism. The first Windows 3.0 applications will get more press — and be judged less critically — than will comparable later products. Be the first.

This advice may seem not only obvious but also useless, because a couple dozen Windows applications were announced six months ago at the Windows rollout. But the get-there-first truism also seems to work on a niche-by-niche basis. I modify my advice. Be the first at something.

For any new platform, two hot markets are developers and early adopters of new technology. Particularly for these groups, a think-globally-act-locally truism applies, if you happen to live in the San Francisco Bay Area or in the Boston area. Consider keeping costs low by spending most of your marketing budget locally. Check the ad rates in the regional computer publications. Talk to the local papers.

Consider the home market. This was what IBM and a lot of others companies did, to their chagrin, in the early 80s, and they're at it again. Maybe this time they're right. According to the September 10 *Business Week* story on the home market, 3.7 million computers a year are bought to be used in the home, and 20 percent of the PS/2s sold are being used at home.

Does all of this actually constitute a distinct market for home software? Beats me. I don't think Tandy has the answer with the recipe program that it's including with its new home machine (which, incidentally, won't run Windows). The *Business Week* piece suggests that home applications aren't much different from work applications: word processing, accounting, budgeting. But if you have a brilliant idea for a home product, you should know that IBM, Apple, Tandy, and a lot of other hardware companies are going to be pushing computers for the home very hard. Since the early 80's snafu almost certainly had to do with ease of use, your Windows application might do very well.

Don't overlook Asymetrix ToolBook as a prototyping tool. Here's a scenario: You write a design prototype using ToolBook. You add core functionality in DLL functions to produce a working prototype. Now you write the real application around the same DLL functions. At this point you actually have two products under development: A full high-performance version, and a stripped-down, slower, but not unacceptable ToolBook version. This gives you not just a product but a (sort of) product line from (sort of) one development cycle.

You evolve these two products away from one another as you develop them, adding functionality to one (removing it from the other) to differentiate them. You put the ToolBook version out first, staking out your market niche, arousing customer and press interest, and getting cash flowing in earlier than you could if you waited to get a *real* application done.

This buys you two legitimate occasions to talk to the press about your product because you will have two product announcements. And when you later release the non-ToolBook *professional edition*, the press and users already understand the concept of the product and the basic functionality, so their attention is naturally drawn to what is different about this product: Improved performance and niftier features.

When considering maintaining Windows 2.1 compatibility, keep in mind that Microsoft is marketing Windows 3.0 primarily against Windows 2.1. The success of Windows 3.0 will, to some extent, depend on how much users disliked Windows 2.1.

When considering whether or not to develop simultaneously for Windows and PM, consider the cost of delaying your Windows release against the size of the OS/2 market in the near future. In the not-so-near future, you can always port a stable Windows version to OS/2. And consider that developing simultaneously for Windows and PM requires that you care about Microsoft's relationship with IBM, whatever that may be this week.

Michael Swaine
editor-at-large

#171 DECEMBER 1990 $3.50 ($4.50 CANADA)

15 YEARS OF POWER PROGRAMMING

Dr. Dobb's JOURNAL

SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

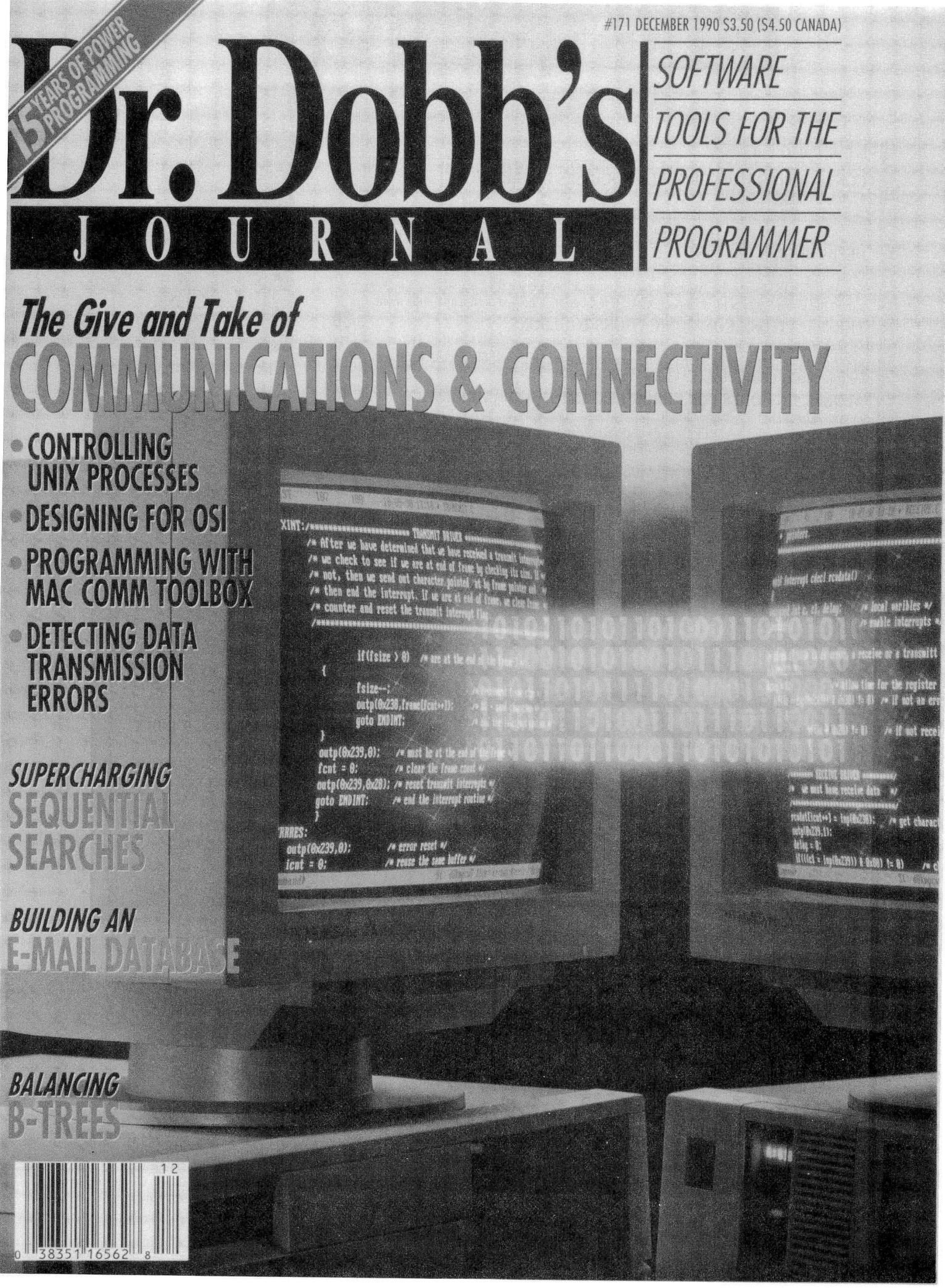
The Give and Take of COMMUNICATIONS & CONNECTIVITY

- CONTROLLING UNIX PROCESSES
- DESIGNING FOR OSI
- PROGRAMMING WITH MAC COMM TOOLBOX
- DETECTING DATA TRANSMISSION ERRORS

SUPERCHARGING SEQUENTIAL SEARCHES

BUILDING AN E-MAIL DATABASE

BALANCING B-TREES



12
0 38351 16562 8

CONTENTS

DECEMBER 1990
VOLUME 15, ISSUE 12

FEATURES

CONTROLLING BACKGROUND PROCESSES UNDER UNIX 16
by Barr E. Bauer
This system, chiefly written in Bourne shell script, "user-izes" the management of background processes that run locally and across a network.

DESIGNING AN OSI TEST BED 24
by Kenneth L. Crocker and Michael T. Thompson
Ken and Michael discuss the synchronous communications device drivers that play a key part in the Open Systems Interconnection (OSI) test bed they helped design and implement.

THE MACINTOSH COMMUNICATIONS TOOLBOX 38
by Don Gaspar
Apple's Communications Toolbox includes connection, terminal, and file transfer managers for constructing dynamic communications applications.

ALGEBRAIC CODES FOR ERROR DETECTION AND CORRECTION 46
by Hsi-Chiu Liu
One of the most efficient methods of error detection and correction is *algebraic coding*, which requires only a minimal amount of bit redundancy in forming code words.

SUPERCHARGING SEQUENTIAL SEARCHES 54
by Walter Williams
Here's a simple algorithm that can speed up a sequential search by a factor of two or more.

EXAMINING ROOM

EXAMINING THE ZINC INTERFACE LIBRARY 64
by Gary Entsminger
The Zinc Interface Library is a C++ class library for constructing graphics- and text-application interfaces.

PROGRAMMER'S WORKBENCH

A DATABASE SYSTEM FOR AUTOMATING E-MAIL 72
by Chris Ohlsen
Chris presents a message storage and retrieval system for electronic mail built around Borland's Paradox database engine.

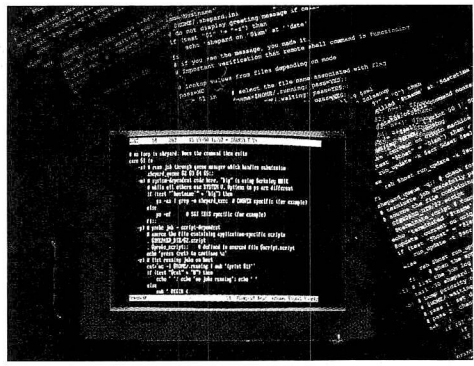
COLUMNS

PROGRAMMING PARADIGMS 113
by Michael Swaine
Michael wraps up loose ends that have come unraveled over the past year.

C PROGRAMMING 121
by Al Stevens
Al examines, updates, and balances a B-tree he first presented last spring.

STRUCTURED PROGRAMMING 131
by Jeff Duntemann
Who knows what evil lurks behind those traditional menu-tree applications? Event-driven architectures may mean you don't even have to ask the question.

PROGRAMMER'S BOOKSHELF 139
by Ray Duncan
If you're starting up or restructuring your business, Ray recommends a small, but important, book called *Peopleware*.



DEPARTMENTS

EDITORIAL 6
by Jonathan Erickson

LETTERS 8
by you

SWAINE'S FLAMES 160
by Michael Swaine

PROGRAMMER'S SERVICES

ADVERTISER INDEX 152
where to go for more information on products

OF INTEREST 153
compiled by Janna Custer

PROGRAMMER'S MARKETPLACE 154
classified ads

SOURCE CODE AVAILABILITY
As a service to our readers, all source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (inside Calif.) or 800-533-4372 (outside Calif.). Specify issue number and disk format. Code is also available through the DDJ Forum on CompuServe (type *GO DDJ*) and through M&T's Telepath Conferencing system.

NEXT ISSUE

Happy Birthday to the Doctor! January 1991 is our 15th anniversary issue and, along with a look at software design, we'll examine topics important to the past and future of computer programming.

Conferences and Contests

Technical editor Ray Valdés recently spent a week in sunny New Jersey attending the "C++ at Work" conference sponsored by the Wang Institute of Boston University and the *C++ Report*. What follows is the short version of his report:

> Now that some of the initial hype has subsided, where does C++ stand today? What are its prospects? Should you switch from whatever language you're using now, or wait and see?
>
> These were some of the questions raised at the "C++ at Work" conference held in Secaucus, New Jersey. The principal advantage of the conference location was its proximity to Bell Labs, cradle of both the C language and its object-oriented descendant. Human languages have a strong cultural component, programming languages much less so. But the C++ language is complex and subtle enough that an immersion into the native culture of that language is necessary in order to use this tool effectively. And there were enough native speakers from Bell Labs at this C++ conference to make the visit worthwhile.
>
> One realization, for me, was how large and complex the language is in comparison to C. For PC programmers moving to C from other languages, making the transition to C++ will be an equally large step. It is paradoxical that, on one hand, you must be both well grounded in the culture of C (pointers, pre- and postoperators, casts, and so on) to fully understand C++. On the other hand, you must be willing to throw out the "bad" procedural-language mind-set in favor of "good" object-oriented worldview.
>
> The clarity of the object-oriented view is sullied by real-world considerations of performance and efficiency. These considerations burden the language with multiple alternative techniques (six different kinds of inheritance, for example) and a need to always keep the underlying machinery in mind. For example, it is hard to read or debug a program without knowing the entire context of a class's ancestors (such as any overloaded operators or overriden methods), as well as knowing how the compiler creates temporaries when it evaluates an expression (for example, which constructors will be invoked).
>
> (As an aside for those who find the language cumbersome or wordy, consider a tour I recently took through a very large software system, composed of hundreds of thousands of lines of C and many megabytes of executable. In the course of making this system manageable, the developers had to concoct many of the object-oriented mechanisms offered by the C++ language. Because these were implemented as stylistic conventions and preprocessor macros, the resulting system is both harder to read and less reliable than a C++ equivalent would be.)
>
> Despite these difficulties, it is clear that the language will be the mainstream development vehicle of the 90s. Not wishing to start any religious wars, I believe that Smalltalk is simpler, cleaner, more powerful, and can exist in a fast enough implementation. Yet C++ will prevail, for reasons that are as much cultural, historical, and business oriented as they are technological.
>
> As evidence, some of the speakers at the conference were émigrés from Lisp (Dick Gabriel, of Lucid Inc.) and Smalltalk (Ted Goldstein, formerly of ParcPlace Systems). They are presently immersed in bringing to C++ some of the nifty advanced tools currently enjoyed only by users of those languages.
>
> As a speaker said at the conference, C++ is a language invented by expert programmers and computer scientists to make their work easier and more productive. In the hands of experts, it is a powerful tool that makes development of large software systems manageable. In less-skilled or inexperienced hands, it can prove quite dangerous.
>
> Bjarne Stroustrop had some useful advice near the end of the week: "Ease yourself into the language. Don't feel you have to use all of the features, and don't try to use them all on the first day."

Calling All Student Programmers

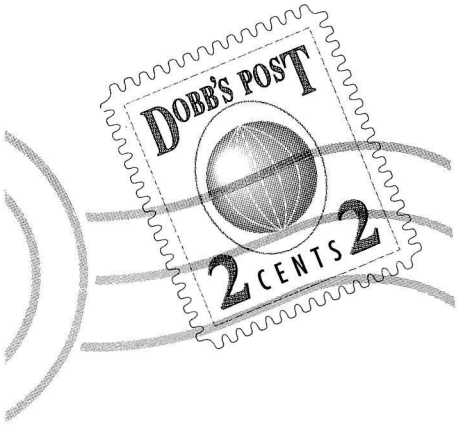
Symantec, makers of Think C, Think Pascal, and a passel of other development tools, is sponsoring a programming contest for high school and college students. The competition consists of programming questions and problems. Individual students or entire classes can enter. There's one version of the contest for high school, another for college.

Two grand prizes will be awarded — one for high school and one for college — and they're not bad. Each grand prize winner will get a $5,000 scholarship, a Macintosh IIci, and a lifetime subscription to *Dr. Dobb's Journal*. Four runners up (two from high school, two from college) will also receive prizes, including a year's subscription to *DDJ*.

Entries will be accepted through March 1991, with prizes awarded in June 1991; judging will be by Symantec engineers. To register or find out more information, teachers or students should call Symantec Contest Hotline, 617-275-4800.



Jonathan Erickson
editor-in-chief



Porting Fortran

Dear *DDJ*,

The article "Porting Fortran Programs from Minis to PCs," by John Bradberry (September 1990) brought back memories of the most thankless task I have ever undertaken. . . porting a moderately sized scientific Fortran program to a PC.

I can still recall staring aghast at the countless travesties in style and practice, marveling at the complete lack of structure. Imagine a seemingly endless printout of unindented, line-numbered code with scores of undeclared variables with ingenious names like dxy15, ixxx, ix, ixx, ix1, ixy, nrtsc, iscale1, etc. Every eight or nine vertical inches of code would be interrupted by a block of seven or eight consecutive computed *if* statements. If subroutines performing multiple tasks are a Do Not, that is still better than a huge program with no subroutines at all . . . just *goto*s and *if* statements returning control somewhere depending upon the value of a global flag defining the original calling context.

It seemed sad somehow that a very useful, even brilliant, solution to an important problem found expression in such an ungainly program. On the bright side, what was once ugly is now beautiful, and what was once Fortran is now C.

Peter Matsunaga
Aiea, Hawaii

Dear *DDJ*,

It was a real pleasure to see the article "Porting Fortran Programs from Minis to PCs," by John Bradberry (September 1990). The table of "Do's and Don'ts" was especially interesting to a veteran of some 30 years of wrestling with Fortran mysteries.

But on to the bad news: The listings accompanying the article reveal a commonly made mistake in dealing with type conversions in computed or replacement statements. Consider, for example, the 45th line of the listing, shown on page 80: PI=3.14159265. The assumption usually made is that the information stored in variable PI will be of double precision since PI was declared as type REAL*8.

Example 1 shows that this is not always true. Many compilers, including the Microsoft Fortran 5.0 used here, do not conform to the assumption. The value actually stored in PI in this example turned out to be 3.1459274. This is nowhere near the 15 or 16 digits of precision expected.

In the absence of cues explicitly designating the right side of a statement as double precision, the compilers apparently evaluate the right side in single precision registers. When the time comes to finally replace the left side variable, the right side result, in a single precision register, is transferred to a double precision register and then to the double precision storage location of the variable.

Having found several instances over the years where taking things for granted got me less than I expected, it has become my practice to make statements involving type conversion as "bullet-proof" as possible. This means things like explicitly putting in the SNGL and DBLE functions wherever those effects are actually intended to be. In the case of constants, such as 3.1459265, the only way to make constants explicitly double precision is to always include a "D" exponent. For example: PI=3.14159265D0.

George Zabriskie
Huntsville, Alabama

John responds: Mr. Zabriskie correctly points out a flaw in the initialization of double precision constants in the Fortran listings. Type conversion and mixed-mode arithmetic present such potential problems (in all languages) and were not adequately addressed in my article due to scope and space limitations. The actual use of the constant(s) had no adverse effect on the demonstrated application, but good programming practices and consistency are always worth pointing out.

And Porting C

Dear *DDJ*,

I just read "Porting C Programs to 386 Protected Mode," by William F. Dudley, Jr. (August 1990) and am writing in response to the statement:

"Now I have a hairy preprocessor macro that computes the size of a 'data element,' which is really the biggest of any of several different structs." (p. 17)

This may not be a complete problem exposition, but it would be unfortunate if such a mechanism were always required. A simple solution is shown in Example 2. This consumes no run-time memory, though if you need the size of a data element, you can now index more efficiently, as in Example 3.

John Deurbrouck
Mountlake Terrace, Washington

10-Key Update

Dear *DDJ*,

Although I found Jeff Duntemann's column in the August 1990 *DDJ* interesting, I must take issue with his choice of user interface. Full-screen editors with mouse control have their place, but entering numbers from a bingo card is not that place.

Have you ever seen someone use a 10-key pad such as the one on the right of the PC's keyboard? A skilled operator has two important qualities: He can operate by touch and he can do so very fast. Requiring a 10-key operator to move his attention between the bingo

(continued on page 12)

```
C:\>TYPE PIE.FOR
      PROGRAM PIE
      REAL*8 PI_ONE, PI_TWO
C
      PI_ONE=3.14159265       !Common method - without exponent
      PI_TWO=3.14159265D0     !Correct method - with D exponent
C
      WRITE (*,100) PI_ONE, PI_ONE
  100 FORMAT(' This is PI stored without a ''D'' designator: ',F12.8,3X,
     .       Z16,/)
      WRITE (*,110) PI_TWO, PI_TWO
  110 FORMAT(' This is PI stored with a ''D'' designator:  ',F12.8,3X,
     .       Z16,/)
      STOP
      END

C:\PIE
This is PI stored without a ''D'' designator: 3.1459274 400921FB60000000
This is PI stored with a ''D'' designator: 3.14159265 400921FB53C8D4F1

Stop - Program terminated.

C:\>
```

Example 1

(continued from page 8)
card and the screen is inefficient; to further require him to translate a number into relative screen coordinates (let's see; I'm on 100 and 205 is also circled — that's five over and two down) is double so.

Back when I did this sort of thing for a living, I could even move my hands between the keypad and the alpha keyboard without looking.

Please take the operator into account when designing user interfaces. Although screen-oriented editors are flashy and fun to program, I believe that Mr. Duntemann's users would appreciate a simpler, enter-the-number interface.

Roger Ivie
Logan, Utah

And More on AND

Dear *DDJ*,

I too have heard about the unusual programming language Michael Swaine discussed at the end of his January 1990 "Programming Paradigms" column. There were several things about this language he did not mention. This language is particularly well-suited for programming cellular automatons. In addition, AND programs are not structured in a top-down fashion, but form a double helix instead.

Even though there are many millions of different kinds of programs written in AND, current research is trying to take parts of several programs and recombine the pieces into new programs. You would think that with all the programs written in AND there would be one that would fulfill any need. Maybe we conventional programmers can learn a lesson from the AND researchers.

AND programs are susceptible to viruses. Strangely enough, some kinds of bugs in AND programs occasionally manifest themselves as another programming language, the one Swaine discussed in the first part of his column [Lisp].

You would think that with all the AND programs around, there would be a debugger. Maybe there are some technical problems that can't be overcome, like the problem of interpreting the object code. But with only three symbols to consider, this shouldn't be very hard.

At any rate, this is what I know about this unusual programming language. I look forward to hearing more about AND. And please find out the correct name of this language. I'd be ever so interested to know.

Clifford V. Moravetz
Gays Mills, Wisconsin

Network Detection

Dear *DDJ*,

Regarding Al Stevens's column about the NetWare API (August 1990), I would like to share a method I use for detecting a network. Whenever you call NetWare's *GetServerInformation* function, the file server name will be zero length if no server can be found. Since I am using C++, I take advantage of default parameters to let you pass the address of a FILE_SERVER_INFO structure (see Example 4) if you so desire, which will then come back containing information on the default server (if any). I think Al will agree this method is a whole bunch simpler.

Now that I've stopped to think of what this function does, it's erroneous to call it "loggedin," since what it actually tells you is whether a file server is detected. Guess I'll have to change its name. . . .

Dave Nelson
Jet Propulsion Laboratory
Pasadena, California

Memory Allocation

Dear *DDJ*,

I thoroughly enjoyed Larry Spencer's article "Debugging Memory Allocation Errors," (August 1990). The technical content was valuable, and, of course, any similarity between the scenarios painted by him and the life of "real programmers" was totally unintentional!

To add to the humor, there is a small unintentional problem in the *mem_Track_free( )* function, which partly defeats the purpose of the function. The simplest correction is to include an additional line, *rc = 0*, at the very end of the code on the left column on page 179.

To those who did not use Larry's code, I suggest you keep it in a safe place (that is, permanently on top of the highest pile); you know not the hour . . .

To those who did, make the above modification and check your program again. Whoooooppppssss!! But the program was perfect yesterday!! OK, OK, OK . . . more coffee . . . dammmn . . . here we go again . . .

Michael Kennedy
Dublin, Ireland

Dear *DDJ*,

In the August 1990 issue, Lawrence Spencer points out some of the problems with dynamic memory allocation in a C/Pascal-like language: "Maybe you freed a pointer that you never allocated" or "Maybe you allocated memory but never freed it" and points out that even veteran programmers are liable to make mistakes of this kind. The results are frequently disastrous, as he points out about the hypothetical program HIGHRISE: "but once in a while it locks up the computer" and "Also, what about the time HIGHRISE forecasted a profit of $17,546,321.97 on a $1,000 investment? . . . Please, let it have been a hardware error!" He then presents some practical solutions for handling this problem in C (I personally have always tended to use something like the solution he presents, in my larger C projects).

But there are other, more general solutions to the problem, albeit outside the context of the article. Use of another language — a language like Lisp,

(continued on page 14)

```
union checksize {
       struct  struct_1;
       struct  struct_2;
       struct  struct_3;
  };
  #define DATA_ELEMENT_SIZE sizeof(union check.size)
```

Example 2

```
struct struct_1 sl_init;
   union check_size aray [40], *ptr;
   int x;
   ptr = array;
   for (x=0; x<40; ptr++, x++)ptr->struct_1 = sl_init;
```

Example 3

```
#include <nit.h>     // the NetWare API header
                     // struct FILE_SERVER_INFO is defined in nit.h
typedef char bool;

bool loggedin(FILE_SERV_INFO * f = NULL){
FILE_SERV_INFO fs, * fp;
     fp = (f==NULL)? &fs : f;      //use the struct passed, if any
     GetServerInformation(sizeof(FILE_SERV_INFO),fp);
     return fp->serverName[0];
     }
```

Example 4

(continued from page 12)
for instance. Unfortunately, the memory management of Lisp can cause performance problems. Not to mention that the venerable Lisp can be a somewhat alien experience for programmers steeped in the C/Pascal tradition.

Somewhere in between is the new language Clu` ("Klu-prime") we provide: This language is a modern version of MIT's CLU language. While memory allocation and deallocation is handled automatically in this language, the performance of this allocation/deallocation is for the most part similar to the programmer making *malloc( )* and *free( )* calls manually. (This however, is not the primary advantage of using Clu`, which supports abstract data types with a very simplified inheritance facility as well as other simple and powerful concepts such as iterators, exception handling, and parametrization.)

In any significantly complex task, dynamic memory allocation/deallocation tends to be enough of a sink of programmer time and effectiveness, and enough source of errors, that I just do not see it remaining under manual control in the programming languages and environments of the future.

Mukesh Prasad
Meta Mind Inc.
East Haven, Connecticutt

Bézier Update

Dear *DDJ*,

I found the July 1990 article "Drawing Character Shapes with Bézier Curves," by Todd King very interesting. Todd is correct in pointing out that the Bézier curve rendering could be improved, especially the parametric equation method. Here is one way to do it.

The equation used for determining the x coordinate on page 48 of the July issue performs fourteen floating-point multiplications, six subtractions, and three additions, plus one to increment the parameter *t*. Of course, the six subtractions can be reduced to a single subtraction by storing the value of $(1 - t)$ in a variable. Actually, this can be simplified further by realizing that the parameter *t* starts at 0.0 and is incremented each time by a set stepping value. This implies that a separate variable can be used for $(1 - t)$, which would be initialized to 1.0 and decremented by the stepping value. The six subtractions are replaced by a parameter that is decremented by a constant only once per iteration!

The equation can be simplified by reducing the number of floating-point multiplications, using Horner's rule:

$$f(t) = [(at + b)t + c]t + d$$

So if $(1 - t)$'s are factored out, the first equation becomes:

$$x(t) = [(x_1(1 - t) + 3tx_2)(1 - t) + 3t^2x_3]$$

$$(1 - t) + t^3 x_4$$

This equation performs eleven multiplications, three subtractions, and three additions. It is possible to factor *t*s instead of $(1 - t)$ terms if that's more convenient. In any case, this demonstrates that the first equation is not minimal and can be improved using simple mathematics. More can probably be done.

Todd has a procedure, *hull*, that shows the convex hull of the four control points. This procedure is incorrect, however. I made a similar mistake in one of my computer graphics projects. For example, if a line was drawn between the first and the last control point, and the second control point was placed on one side of that line, whereas the third control point was placed on the other side of the line, the hull that Todd's procedure draws would not be convex and would not contain the Bézier curve. One of the reasons why folks determine convex hulls is for clipping. If the convex hull of a curve is inside a window, then the whole curve must be inside the window. This means that each of the points on the curve does not need to be checked to see if it is inside the window, and that only the control points need to be tested. Dr. Sedgewick devotes a whole chapter on convex hull determination in his book *Algorithms* (Addison-Wesley, 1988). It's a very good reference on many topics.

Victor J. Duvanenko
Knightdale, North Carolina

We welcome your comments (and suggestions). Mail your letters (include disk if your letter is lengthy or contains code) to DDJ, *501 Galveston Dr., Redwood City, CA 94063, or send them electronically to CompuServe 76704,50 or via MCI Mail, c/o* DDJ. *Please include your name, city, and state. We reserve the right to edit letters.*

DDJ

STATEMENT OF OWNERSHIP, MANAGEMENT, AND CIRCULATION

1. Title of publication: *Dr. Dobb's Journal.* Publication No. 1044789X.
2. Date of filing: October 1, 1990.
3. Frequency of issue: Monthly (12 issues, $29.97).
4. Number of issues published annually: 12.
5. Complete mailing address of known office of publication: 501 Galveston Dr., Redwood City, San Mateo County, CA 94063.
6. Complete mailing address of the headquarters of general business offices of the publisher: 501 Galveston Dr., Redwood City, CA 94063.
7. Names and addresses of publisher, editor, and managing editor: publisher, Peter Hutchinson, 501 Galveston Dr., Redwood City, CA 94063; editor, Jonathan Erickson, 501 Galveston Dr., Redwood City, CA 94063; managing editor, Monica E. Berg, 501 Galveston Dr., Redwood City, CA 94063.
8. Owner: M&T Publishing, Inc., 501 Galveston Dr., Redwood City, CA 94063.
9. Known bondholders, mortgages, and other security holders owning or holding 1 percent or more of total amount of bonds, mortgages, or other securities: Markt & Technik Verlag Aktiengeselltschaft, Hans-Pinsel-Strasse 2, 8013 Haar bei Munich, Germany.
10. Extent and nature of circulation:
    A. Total number of copies (net press run). Average number of copies each issue during preceding 12 months: 122,447. Actual number of copies of single issue published nearest to filing date: 126,646.
    B. Paid and/or requested circulation: 1. Sales through dealers and carriers, street vendors, and counter sales. Average number of copies each issue during preceding 12 months: 21,157. Actual number of copies of single issue published nearest to filing date: 21,669. 2. Mail subscription. Average number of copies each issue during preceding 12 months: 80,894. Actual number of copies of single issue published nearest to filing date: 83,489.
    C. Total paid and/or requested circulation. Average number of copies each issue during preceding 12 months: 102,051. Actual number of copies of single issue published nearest to filing date: 105,158.
    D. Free distribution by mail, carrier, or other means, samples, complimentary, and other free copies. Average number of copies each issue during preceding 12 months: 3,566. Actual number of copies of single issue published nearest to filing date: 3,957.
    E. Total distribution. Average number of copies each issue during preceding 12 months: 105,617. Actual number of copies of single issue published nearest to filing date: 109,115.
    F. Copies not distributed 1. Office use, left over, unaccounted, spoiled after printing. Average number of copies each issue during preceding 12 months: 792. Actual number of copies of single issue published nearest to filing date: 1,075. Return from news agents. Average number of copies each issue during preceding 12 months: 16,038. Actual number of copies of single issue published nearest to filing date: 16,456.
    G. Total. Average number of copies each issue during preceding 12 months: 122,447. Actual number of copies of single issue published nearest to filing date: 126,646.

I certify that the statements made by me above are correct and complete.

Peter Hutchinson, Publisher

Controlling Background Processes Under Unix

Here's a system that "user-izes" process management

Barr E. Bauer

undertake scientific computing using a number of networked workstations, compute servers, and a supercomputer. Typically, problems are set up and the results are analyzed on the workstations, while the actual number crunching activity is performed on the more powerful machines. The description of this approach is straightforward, but the tools necessary to manage the jobs that originate from a central workstation, and are then run on a variety of remote hosts over a network, are not generally available.

For instance, with the standard available Unix System V tools, you cannot ask user-oriented questions, such as "What applications and datasets are running on any remote host?" "Which applications are pending?" or "What is the status of a certain job?" without resorting to Unix commands that oftentimes do not look like the user's actual query.

This article presents a system designed to "user-ize" the management of background processes that run both locally and across a network. The system, which I call "shepard," includes a standardized interface that utilizes menus where appropriate to reduce the management of processes down to the essentials: Tasks, executable scripts, dataset names, and remote host names. The specifics of the process of executing, monitoring, and retrieving programs that run remotely are hidden in shell scripts. The bulk of this system is written in Bourne shell script and is reasonably portable.

System Overview

The basic idea behind this system is to use a series of scripts to control all activities that occur locally or on a remote host, and to do so in a simple, straightforward, user-oriented manner. Integral to this activity is the establishment of a queuing system to handle job control, simplify the selection of the host and programs, invoke the programs, move the data files over a network, and record all activities. The core of the system consists of five Bourne shell scripts — *run*, *shepard*, *shepard_ queue*, *shepard_exec*, and *run_update* — and a number of application-specific control files that contain either tables or control scripts. When the system runs it generates a number of status files to show the jobs that are waiting, running, finished, and restartable on each remote host. The system also generates a file on the origin machine to show the status and location of all jobs.

I have commented the code extensively to describe its features and operation. Descriptions of the scripts are presented here to give you a general understanding of the role of each script in the overall system. For specifics, refer to the listings.

The script *run* is a menu-driven task manager that queries for a task, a script, a dataset, and the host. *run* then carries out the requested task by invoking *shepard* on the selected host through a remote shell (*rsh*) command. *run* maintains a list of jobs originating from the origin machine as well as a log file, and serves as a hub that coordinates jobs run on several machines. Through *run*, the user knows which programs are using which datasets and where those programs are running. The tasks managed by *run* are simple and include starting and stopping the execution of programs, determining the status of jobs running on a specific remote host, and probing running jobs for intermediate results or status information. *run* performs all of its work, including the generation of most menus, through the other scripts across the network by use of *rsh* calls.

The types of tasks that the shepard system currently performs include:

- Execute a job, locally or over a network by a remote host
- Monitor the remote host
- Probe a running job for intermediate results
- Kill a running job with extreme prejudice
- Halt a running job in a controlled manner
- Restart a job

Barr works for Schering-Plough Research, a pharmaceutical company. He can be reached at 60 Orange Street, Bloomfield, NJ 07003.

- List running, waiting (enqueued), restartable, and finished jobs
- List general and error log files

The script *shepard* performs all of the process management for a specific host. *shepard* takes as its arguments the task, the script, the dataset, the origin machine, and the dataset directory. Execution is handled by *shepard_queue*. The process of updating the status file on the origin machine is performed by *run_update*, and all other activities are conducted directly in *shepard*. Lists of running and waiting jobs, as well as an execution log are maintained and displayed when requested by *run*. The process of adjusting the control files, and updating the status file on the origin machine when a job is completed is also handled in *shepard*.

The script *shepard_queue* maintains a FIFO queuing system that regulates both the numbers and the types of jobs allowed to execute concurrently. This approach is a good way to maintain a balance between throughput and maximum system performance. The executing task is started by invoking *shepard_exec* as a background task. An error log is maintained. Instead of using multiple queue files, a single queue file is used both for the sake of simplicity and because the queue file is used to generate a menu.

The script *shepard_exec* has embedded in it the specifics for moving data into the execution environment, running the selected program, moving the results back, and calling *shepard* to update the control files. The specifics of program invocation — the process of copying data files from origin to host, executing the program, and transferring the result files back to the origin machine — are hidden in scripts. *shepard_exec* needs only the name of the script and the specific dataset for the run; all else is controlled by the specifics in the scripts.

The script *run_update* updates the status file on the origin machine to reflect the current status of the job. *run_update* is invoked by remote shell calls from both *shepard* and *shepard_queue*.

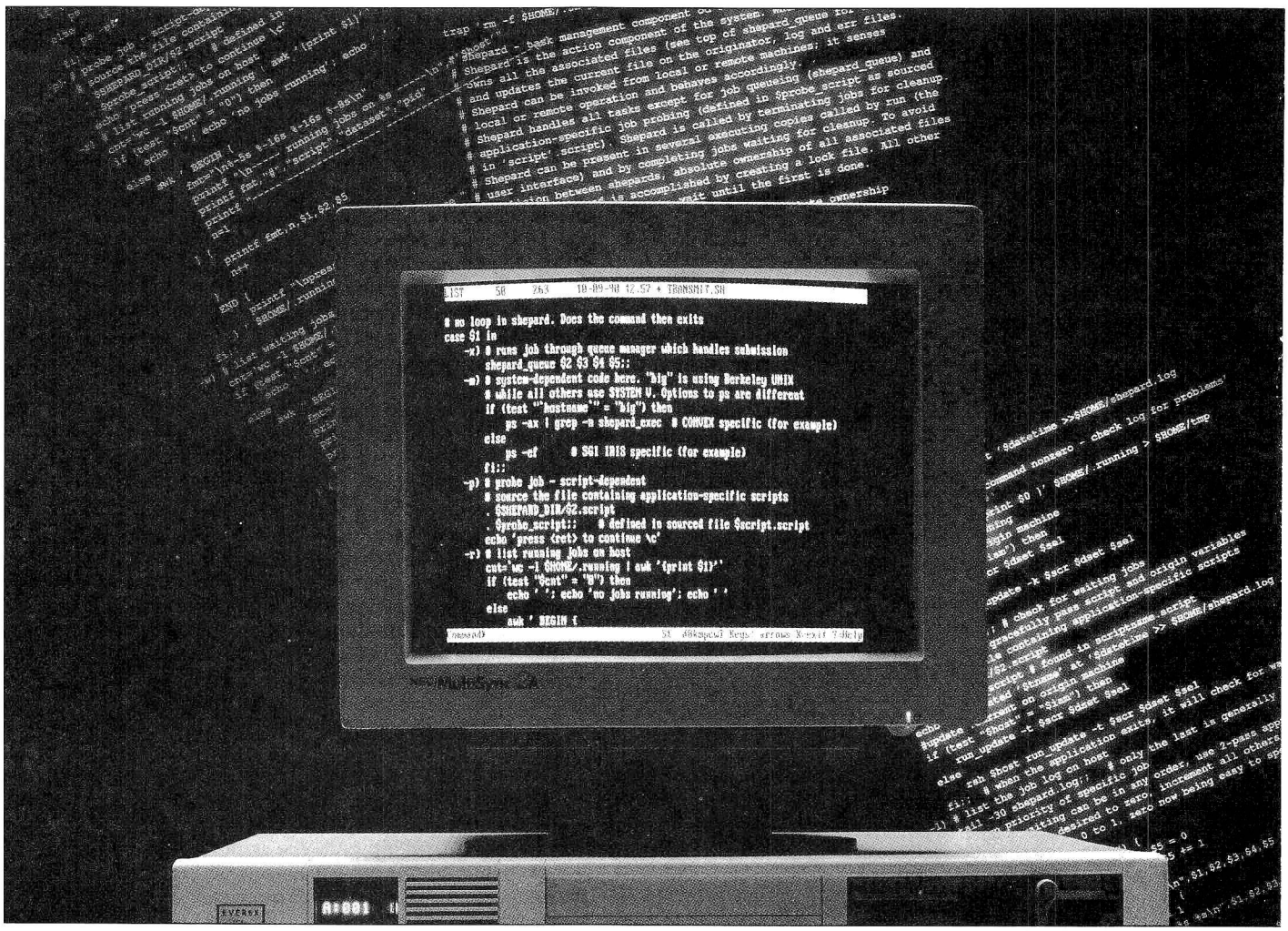
Networking

Three types of data transfer are supported: *remote*, *nfs*, and *server*. The best choice depends very much upon your particular environment and operating preferences. The choice of network is set in *.shepard.ini*.

In *remote* mode, the data is copied from its directory on the origin machine into the user's login account on the selected host and then executed. The output is transferred back to the origin. The input and the output exist on both machines. This approach provides a more robust operating mode when the network is unreliable, but it is not the best mode to use with very large datasets.

In *nfs* mode, the data is accessed through remotely mounted network drives by using Sun's NFS Network File System, and is not actually moved. The output in *nfs* mode is written directly to the network drive. Only one copy of the input and the output is generated. This mode is sensitive to network crashes.

In *server* mode, the data exists, and the execution is performed, entirely on the compute server. Only one copy of the input and the output is generated. This is the best



mode to use with large datasets, and is also useful when the network is unreliable because the data never moves over the network.

I generally use the *server* mode through an NFS-mounted directory. This method provides the benefits of direct access to data without having to move it, and is especially beneficial when the datasets are large.

An Example

The control files in this example are those used in my working environment, and they reflect both the machines and the programs I use in computational medicinal chemistry. The application example is for a version of BATCHMIN (Columbia University), which is the premiere program for molecular simulation. The application is dimensioned to handle protein-sized problems, and is run only on a Convex supercomputer. The problems are set up graphically on an Iris workstation and then are transferred to the Convex for simulation. (The simulation process can last longer than a week for a single problem.) The application script is called *bmin31lv*, and the dataset is referred to as *bmintest*.

The operation of the user interface is described in the comments. The selection of executable scripts, the performance of status queries for running jobs and the like, and the execution of available tasks are all menu-driven activities. I have included niceties such as the listings of completed jobs when *run* is started, plus the retention of selected values for scripts, datasets, and hosts as defaults for the next invocation of *run*.

When a job is intended for launching, *run* passes the selection information to *shepard* on the selected host, and then adds the new job to the origin machine's *.current* file with status *STARTED*. At this point *run* is no longer involved with the process.

When it's invoked by *run*, *shepard* creates a lock file, *.sheplock*. *.sheplock* forces all other invocations of *shepard* to wait and also insures sole access of all control files by *shepard*. *.sheplock*'s control extends through the invocation of *shepard_queue*. *shepard* passes the information to *shepard_queue*, which places the job in the *.waiting* file and gives the job the highest priority value for that script so that the newly submitted job will be the last one to execute. The origin machine's *.current* file is updated to status *WAITING*.

shepard_queue checks the job count of a particular script against the maximum job count in *.limits* and proceeds to execution if below, or exits, leaving the job enqueued. To execute the job, the script *shepard_exec* is passed all the relevant job information and submitted as a background process. The job is removed from *.waiting* and placed in *.running* with the process ID (pid) by *shepard_exec*. If the job fails it is placed in *.restart*. The *.current* file on the origin machine is updated to status *RUNNING*. *shepard_queue* returns to *shepard*, removes the lock, and exits. A log of all activities is maintained so that the user can later determine what happened.

shepard_queue sources an application-specific file, which in this case is *bmin31lv.script*. This file defines the generic shell variables in *shepard_exec* for the specifics of the application. (The use of shell variables to represent the various file extensions makes the basic *shepard_exec* more flexible.) Data is moved by *bmin31lv_getdata*, which is defined in *$getdata_script*. The application is executed with the assumption that it needs a standard input, and generates output to standard output and standard error. *bmin31lv* takes its command input via the standard input from *bmintest.com*, and generates run output via standard output to *bmintest.log*. It also uses *bmintest.dat*, which is moved along with *bmintest.com* into the execution environment.

(continued from page 18)
bmin31lv also generates a *bmintest.out*, which is returned to the origin machine. The output files are moved back across the network using the script *bmin31lv.putdata*, which is defined in *$putdata_script*. From the user's point of view, all of the requisite files are named, created, and moved based only on the application script and the dataset name; the specifics are buried in the scripts.

Upon completion of this process, *shepard_exec* calls *shepard* with the *-z* option. *shepard_exec* may have to wait while other processes (such as *run*) and other completing jobs execute *shepard*. Once in control, the job is removed from *.running* and placed in *.finished*. The origin machine's *.current* is updated to *DONE*, the log files are updated, and the job passes into history. *shepard_queue* is invoked with the *-q* option to check for waiting jobs, and launch if possible. The final act is to remove *.sheplock*.

A different example is the killing of a running job. When the kill task is selected in *run*, *shepard* is called on the selected host. Once control is established, a menu of running jobs is generated from *.running* and control is returned to *run*. The job to be killed is selected by its entry number. *shepard* is again invoked, and now passes the selection number of the job as the fifth argument. *shepard* looks up the pid of the selected job, and kills the script associated with that pid. The process listing is searched for child processes (which in this case is the application) and the child processes are also killed. The job is removed from *.running* and the log files are updated. Control passes back to *run*.

From the user's point of view, the shepard system achieves its goals of simplifying the network management of jobs, and tells the user which jobs are located where, based upon what is important: the application and the dataset. A benefit of the queue system is that numbers of jobs can be left pending (and easily managed) without fear of "choking" the system if they're improperly enqueued or if they're blocking other types of jobs.

Control Files and Menu Generation

The control files are doubly useful both for generating menus and building tables of values for lookup. In fact, the Unix utility program Awk is handy for both tasks. I provide code examples to show the process of generating "pick-an-item" menus and lookup of values based upon the selected item number. Although, the code examples are very simple, I have never seen Awk used in this manner.

The control files use space separation between the fields. Awk has the flexibility to use selected fields (words) out of a single record (line). This flexibility offers the option to use the first several fields on a line as lookup values, and to use the rest of the fields on the line as information displayed in the menu. An example of this method is *.hosts*, in which the first word on each line represents a possible host, and the remainder of the line describes the host.

File	Format
.runscripts:	hostname script (remainder is descriptive comment)
.hosts:	hostname (remainder is descriptive comment on host)
.tasklist:	flag (remainder is description of 'run' task)
.limits:	hostname script maximum-jobs-per-script
.current:	script dataset host datadir start finish status
.waiting:	script dataset host datadir queue-position
.running:	script dataset host datadir process-id
.finished:	script dataset host datadir finished-time
.restart:	script dataset host datadir

Table 1: *Control file formats*

The formats of the various control files are shown in Table 1. The other control files (*.run.ini* and *.shepard.ini*) and the application-specific scripts are all Bourne shell scripts, and are commented with respect to their functions.

Installation

The scripts and the control files should be placed on all desired machines. Give the scripts *run*, *shepard*, *shepard_queue*, *shepard_exec*, and *run_update* (Listings One through Five, pages 82–88) execution privilege, and place them in a directory accessible through the path. Place the control files for the scripts *run* (*.run.ini*, *.current*, *.hosts*, *.tasklist*, and *.runscripts*, Listings Six through Nine, page 90) and *shepard* (*.limits* and *.shepard.ini*, Listings Ten and Eleven, page 90) in the user's login directory. Create a directory for the application-specific scripts (Listings Twelve through Sixteen, page 90) and move those scripts into the directory. Finally, compile lockon.c (Listing Seventeen, page 90) and place the executable in a directory accessible through the path.

Modify the control files according to your needs and your network setup. Change *.hosts* to reflect the accessible machines on your network, and modify *.runscripts* so that it uses both the accessible machines and the control scripts for your applications. Make the same changes to *.limits* that is made to *.runscripts*, and add the numbers that correspond to a desired mix of running jobs for a maximum load situation.

Create the application-specific scripts for your specific applications and needs. Use the example scripts for *bmin31lv* as a template. If your applications do not have a graceful terminate or probe capability, set the variable assignments in the master application *.script* file equal to no strings.

Modify the control file *.shepard.ini* to reflect your applications present on all available platforms, your preferred mode of network access, and the directory in which the application-specific scripts are kept. (The application names are without the *.script* extension.) Make sure that the network mode is the same in the *.shepard.ini* files for pairs of platforms. Generally, you cannot mix network modes between specific pairs of machines, although other modes can be selected for other pairs.

Finally, if the machines on your network are not communicating freely, make the necessary changes to the system files that control access. You should have accounts on all desired machines. The */etc/hosts* and */etc/hosts.equiv* on each of a pair of desired machines must be set so that the *rsh* command works properly. If you intend to use *nfs* to move files between files, make sure that your *nfs* is enabled and that the necessary directories are mounted remotely.

The system works without modifications between Silicon Graphics Iris platforms that use Unix System V, Version 3.2, and a Convex C-220 that uses Berkeley Unix 4.2, when all of the communications requirements are met.

Future Enhancements

The shepard system is functional, but not complete. A facility to check for crashed jobs and then clean them up is under development. Also, the changes necessary in order to support multiple users are straightforward, but they introduce a complication with respect to the queuing limits: The mix of jobs must also include a mix of users.

DDJ

(Listings begin on page 82.)

Vote for your favorite feature/article.
Circle Reader Service **No. 1.**

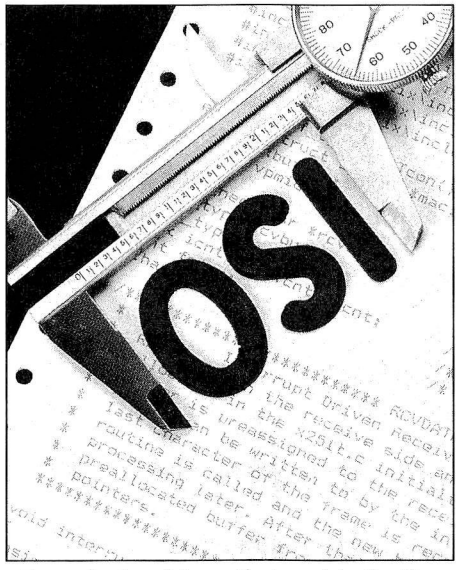
Designing an OSI Test Bed

Synchronous communications device drivers are critical to success

Kenneth L. Crocker and Michael T. Thompson

As of August 1990, new government procurements must comply with the Government Open Systems Interconnection Profile (GOSIP, see Federal Information Processing Standard 146). One outgrowth of this requirement will certainly be accelerated development efforts for portable OSI applications, like the PC-based OSI protocol test bed we describe in this article.

This test bed utilizes commercially available, portable source code to provide a Class Four Transport over X.25 Wide Area Network (WAN) implementation. The Class Four transport service provides connection-oriented, end-to-end service regardless of the underlying network services. The test bed software consists of approximately 42,000 lines of portable C source code purchased from Retix, 8000 lines of C developed to emulate an FAA weather data transfer application, and 200 lines of C to interface the Intel 82530 communications hardware to the Retix software. In particular, we'll focus on developing the synchronous communications device drivers that enable the Retix software to communicate with the 82530 hardware.



When designing the hardware-level device driver, you must address: 1. The amount of time allocated to input/output (I/O) routines based upon the highest expected data rate; 2. Timing requirements unique to the communications hardware; and 3. The interface to the portable code. The information presented in this article can be used as an example of how to design an interface to the Intel 82530 controller, as well as an example of overall system design when using portable communications software.

Ken is a member of the technical staff at The MITRE Corp. and Michael is a senior software engineer at Planning Systems Inc. They can be reached at 7525 Colshire Dr., Mailstop W389, McLean, VA 22102. Ken and Michael can also be reached via e-mail at kcrocker@mitre.org and mw3075@mitre.org, respectively.

Software and Hardware Platform

The test bed itself is a DOS-based system using Microsoft C 5.1 and Microsoft Assembler (MASM) 5.0 as the development environment. A non-Unix operating system was chosen to better study portability issues. Figure 1 shows the communications and software architecture currently implemented on the test bed. As shown in the figure, Class Four Transport, X.25 packet layer protocol (PLP), and Link Access Procedure Balanced (LAPB) were purchased as portable source code from Retix. Operating as a transport user, emulations of the FAA Data Link Processor (DLP) and Weather Message Switching Center Replacement (WMSCR) provide realistic FAA weather data to demonstrate the OSI protocols. The device driver forms the interface between the Intel 82530 Serial Communications Controller (SCC) and the lower interface of the Retix LAPB module. The SCC is implemented on an IBM PC/XT-compatible communications peripheral purchased from Sealevel Systems. In our implementation, the 82530 provides limited link-layer functions (for example, framing and error checking) with RS-232 drivers provided by the Sealevel Systems physical interface. The current end-state hardware platform of the test bed is the IBM PC/XT 286. The XT 286 operates at a clock speed of 6 MHz with zero wait states. As a development platform, the Compaq 386/33 is

(continued on page 28)

(continued from page 24)
used in conjunction with a software in-circuit emulation package. The timing analysis explained in this article is based upon the XT 286 platform.

Device Driver Timing Requirements

Unix, a multiuser, multitasking operating system (OS), was the OS Retix chose to develop its portable OSI software products in. MS-DOS, a single-tasking OS, allows one program or process to run at any given time. This means that the device driver can either move data via software polling or hardware interrupts. In polling, the device driver routines are placed inside what is essentially an infinite loop. The polling routine is then executed during each pass through the loop; however, every instruction that is added to the polling code, such as program diagnostics, increases the total loop execution time, which means that the communications hardware is interrogated less often. Conversely, interrupt- or event-driven communications alert the processor whenever data has been received or the transmit buffer is ready to accept more data. The interrupt-driven solution gives program control to the device driver whenever it is required, unlike the polling method, where program control is gained on a per-loop basis. The system described in this article uses a combination of polled and interrupt-driven I/O. The upper layer portable code is polled to check a series of event queues, while the receive and transmit data are handled via interrupts. The design of the interrupt handler for this interface must account for the following timing considerations: 1. The overall time to execute the interrupt routine must allow for non-interrupt processing to occur during data transfer; 2. The cycle and reset recovery times of the SCC must be met; and 3. The timing window of the SCC for reading and writing data must also be met.

Because the 9600 bit-per-second (bps) inbound data will generate a receive interrupt every 104 microseconds (µs) in the test bed environment, the SCC is programmed, as will be described later, to generate an interrupt for every 8 bits of received data and whenever the transmit buffer is empty. The receive interrupt period is derived as:

period of receive interrupt = (no. bits per character from SCC) / (transmission rate in bps)

The 104-µs time span translates into approximately 624 timing, or clock, states for the 6-MHz XT 286. The 80286 can process approximately 100 assembler instructions in that time frame, assuming an average of six timing states per instruction for the types of instructions used in the device driver (for example, *push, pop, mov, test, in, out*) and ignoring other pending interrupts, system calls, and so on. For efficiency, the interrupt routine that loads the incoming data into the communication stack buffer manager should not occupy more than 25 percent of the total interrupt period. In a true full-duplex environment, one-half of the total processing time would be spent handling I/O while the other half would be used by application and communications stack-processing needs. Development in the 6-MHz XT-286 environment has shown that a faster processor is required for this type of interface. At 6 MHz, our interrupt routines can take up to a full character time (or 104 µs) to execute. This is due to the buffer management requirements imposed by the portable software design and the required handshaking with the communications hardware. In our prototype environment, we control the overall throughput of the system by inserting delay to limit the transmit frame rate at the transport layer. In this manner we are able to adjust the transmit throughput at the link layer to accommodate our specific needs. Because LAPB is a windowing, asynchronous, acknowledgment-based protocol, when system A is transmitting a frame to system B, system B will wait until a set number of frames have been correctly received before acknowledging the window to system A. Once a link-layer connection has been established, normal communications continue in this manner.

Assuming a frame size of 512 bytes, this will yield 53 milliseconds (ms) of interrupt processing time and 947 ms to process the frame at the upper layers. For proper throughput efficiency, either a faster processor or a more streamlined interface to the portable code will be required; however, we have been able to successfully interoperate with hardware and software manufactured by different vendors using this processing method.

The second set of timing considerations of the device driver are the cycle and reset recovery times of the SCC. The cycle recovery time applies to the time period between any read or write cycles to the SCC. This time period is defined to be six Processor Clock (PCLK) cycles. The reset recovery time applies to the time required to perform a software or hardware reset. The local PCLK on the Sealevel Systems board operates at 4.9152 MHz. This implies a cycle recovery time of 1.22 µs and a reset recovery time of 2.24 µs. To ensure that this critical timing requirement is met, the resulting assembler output from the C compiler must be analyzed. In sections where the cycle and reset recovery times are not met, dummy code (for example, *nop,jmp$+2*) is inserted to achieve the proper delay. The modified assembler code is then com-

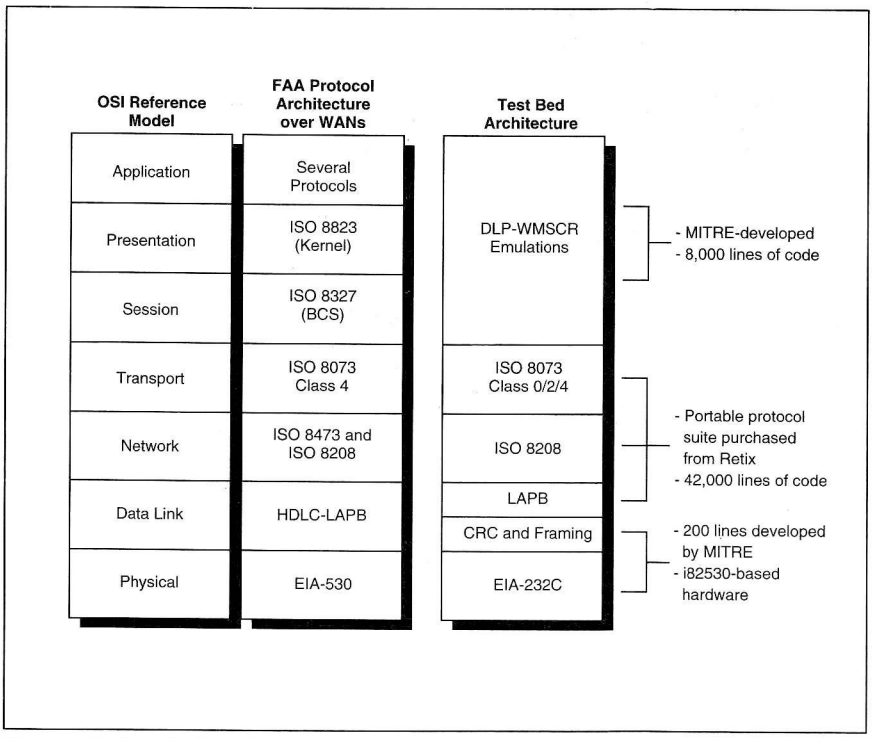


Figure 1: *Test bed architecture*

Dr. Dobb's JOURNAL
SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

PUBLISHER *Peter Hutchinson*

EDITORIAL
EDITOR-IN-CHIEF *Jonathan Erickson*
MANAGING EDITOR *Monica E. Berg*
TECHNICAL EDITORS *Michael Floyd, Ray Valdes, David Betz*
ASSOCIATE MANAGING EDITOR *Janna Custer*
PRODUCTION EDITOR *Tami Zemel*
CONTRIBUTING EDITORS *Al Stevens, Jeff Duntemann, Tom Genereaux, Andrew Schulman, Ray Duncan*
COPY EDITORS *Pamela Dillebay, Nan Fornal*
EDITOR-AT-LARGE *Michael Swaine*

ART/PRODUCTION
ART/PRODUCTION DIRECTOR *Larry L. Clay*
ART DIRECTOR *Michael Hollister*
PRODUCTION SUPERVISOR *Amy Shulman Lesovoy*
TYPOGRAPHERS *Charlene Carpentier, Margaret Anderson*
COVER PHOTOGRAPHER *Michael Carr*

CIRCULATION
DIRECTOR OF CIRCULATION *Maureen Kaminski*
CIRCULATION MANAGER *Randy Robertson*
CIRCULATION PLANNING MANAGER *Manny Sawit*
DIRECT MARKETING MANAGER *Francesca Davies*
SINGLE COPY SALES DIRECTOR *Sarah Forsman*
FULFILLMENT MANAGER *Anne Jean*
DIRECT MARKETING COORDINATOR *Susan Bauman*
PROMOTION COORDINATOR *Philip Tsang*

ADMINISTRATION
VICE PRESIDENT OF FINANCE *Kate Deschamps*
CONTROLLER *Mary Collopy*
CREDIT MANAGER *Betty Arsene*
ACCOUNTING SUPERVISOR *Renate Kernke*
ACCOUNTS RECEIVABLE *Wendy Ho*
ACCOUNTS PAYABLE *LuAnn Rocklewitz*

MARKETING/ADVERTISING
ASSOCIATE PUBLISHER *Karla Spormann*
ADVERTISING COORDINATOR *Laura Stack Pullen*
MARKETING ASSISTANT *Sara Noah Ruddy*
ACCOUNT MANAGERS *see page 152*

M&T PUBLISHING INC.
CHAIRMAN OF THE BOARD *Otmar Weber*
DIRECTOR *C. F. von Quadt*
PRESIDENT *Laird Foshay*
VICE PRESIDENT OF PUBLISHING *William P. Howard*

DR. DOBB'S JOURNAL (USPS 307690) is published monthly by M&T Publishing, Inc., 501 Galveston Dr., Redwood City, CA 94063; 415-366-3600. Second-class postage paid at Redwood City and at additional entry points.

ARTICLE SUBMISSIONS: Send manuscripts and disk (with article, listings, and letter to the editor) to the associate managing editor 415-366-3600.

***DDJ* ON COMPUSERVE:** Type GO DDJ.

SUBSCRIPTION: $29.97 for 1 year; $56.97 for 2 years. Foreign orders must be prepaid in U.S. funds drawn on a U.S. bank. Canada and Mexico: $45.00 per year. All other foreign: $70.00 per year.

POSTMASTER: Send address changes to *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. **ISSN 1044-789X**

CUSTOMER SERVICE: For subscription questions, call toll-free 800-456-1215 (U.S. and Canada). For subscription orders or change of address call 303-447-9330 (all other countries) or write *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. For book/software orders call 800-533-4372 (in California 800-356-2002).

FOREIGN NEWSSTAND DISTRIBUTOR: Worldwide Media Service Inc., 115 E. 23rd St., New York, New York 10010; 212-420-0588. FAX 415-364-8630.

Entire contents copyright ©1990 by M&T Publishing, Inc., unless otherwise noted on specific articles. All rights reserved.

The Audit Bureau

(continued from page 28)

piled using Microsoft MASM 5.0.

The third device driver timing consideration is the maximum time allowed to transfer data to and from the chip. Synchronous communication transfers data in a steady stream. Unlike asynchronous protocols, which use start and stop bits to frame each character, synchronous data is transferred in streams of 128, 256, and greater bytes of data. These data streams, or frames, are separated by flags. Because each byte of data must be transmitted in succession with the previous byte, the transmit routine must place the outbound data into the SCC before the previous byte has been completely transmitted. This timing requirement is imposed by the internal architecture of the 82530. The architecture of the controller and the required interface software are discussed in following sections.

Hardware Architecture and Initialization

The SCC has two independent full-duplex channels with 14 write registers and 7 read registers per channel. All modes of communications are established by the bit values sent to the write registers when the system is initialized. As data is received or transmitted, the read register values will change according to the mode programmed in the original setup. These changes to the read registers can cause hardware or software actions that lead to changes to other registers or other software or hardware reactions.

The SCC also utilizes a 3-byte first-in-first-out (FIFO) receive queue and a 20-bit transmit shift register which is loaded by a 1-byte transmit data buffer. The user of the SCC communicates with both mechanisms via a 1-byte I/O buffer. For synchronous communications, the transmit shift register must always contain data during a frame transmission. If a transmit underrun occurs, the SCC appends the 2-byte checksum field and flag information to the data stream. The receiving side of this data transfer will then see an incomplete link-layer data frame, resulting in a link-layer reject and possibly network-layer error recovery. If the 3-byte receive queue overflows, then receive data is lost, again causing link-layer rejects to occur. As stated in the previous section, the hardware driver must assure that the transmit shift register always contains data when sending a frame and that the receive queue is maintained at a near-empty level.

The initialization of the SCC for interrupt-driven synchronous communications is divided into three parts. Each part is unique to the initialization process and must follow the proper initialization sequence for correct operation of the SCC. The proper initialization sequence is the most critical part of the initialization process. As we learned, the SCC can appear to operate correctly, but not be capable of interoperating with other communications hardware. In our case, the two back-to-back SCCs on our test bed were moving data without errors, yet when we tried to communicate across a packet switch, every frame was rejected because of a bad checksum. The transmitter and receiver were inverting the checksum due to an out-of-sequence, but otherwise correct, initialization process.

During early debugging and experimentation with the device driver, we inserted an initialization sequence that corrected our current problem and caused another problem that did not surface until we introduced a different system, such as the packet switch. The initialization of the SCC must strictly follow the sequence explained in the technical manual, and any "debugging" changes should be checked against this sequence to avoid side effects similar to what we encountered.

The first part of the initialization sequence consists of programming the operation modes of the SCC such as bits-per-character and time constants. As you can see in the source code listings, we begin by forcing a hardware reset. The Synchronous Data Link Control (SDLC) mode of operation is then selected. The SCC actually implements a small subset of the SDLC protocol. This subset, in conjunction with the LAPB functions provided by the Retix link-layer software, meets the link-layer requirements of the test bed. The receive and transmit character lengths are then set to 8 bits each, and 7Eh is selected as the flag pattern to be sent between frames. The initial checksum value is set to FFh (required for proper SDLC operation), the RTxC pin of the 82530 is selected as the source for DTE clocking, and the baud-rate time constant is loaded during the final initialization steps of this phase of SCC initialization.

The second phase of SCC initialization consists of enabling the baud-rate generator to provide transmit clock and enabling the transmitter and receiver. During each subsequent phase of initialization, the previously set values (such as transmit and receive character length) must again be set during each use of a particular write register. As an example, write register 3 is used in this section to enable the receiver, yet all previously programmed information for

(continued from page 30)
write register 3 must be also be included. This phase of initialization is completed by enabling the transmit checksum generator, transmit interrupts, and receive interrupts.

Part three consists of enabling the different interrupts used by the system. In our implementation, the transmit underrun interrupt is enabled and all previously selected interrupts are enabled. This completes the initialization of the SCC. For further initialization information, the reader should consult the 82530 technical reference manual. The remainder of this article describes the software required to interface the Retix LAPB portable software to the SCC, as well as general information concerning the implementation of portable communications software.

Data Buffers Initialization

At the start of the initialization process, a large block of memory is assigned to the system buffer pool so that data buffers can be allocated for both the transmit and receive routines. Some of those buffers are preallocated for use by the receive routine only and put into the *Baddr* array for later use by the receiver. The remaining buffer pool can then be allocated and deallocated to buffers as needed by the transmit routine. After the buffer pool is created, the *init_rcvbuf( )* routine in x251t.c (see Listing One, page 92) is used to fill the receive buffer array with the pointers to the allocated buffers.

Preallocation of the receive buffers is necessary in the synchronous, interrupt-driven environment. Buffers must be allocated when the interrupts are enabled so that if no buffer is currently available, the allocation routine can loop until one becomes available. If interrupts were disabled during this period, no buffers would ever become available — none could ever be freed without the transfer of data.

Receiver

After the system has preallocated the first buffer in the *Baddr* array, the first buffer is then available to be written to by the interrupt receiver routine as soon as the interrupts are enabled. As each new character is received, the pointer to the next memory location is incremented and the character is stored in the buffer. When the last character of the frame is received, the frame is checked for a valid checksum. If the checksum is invalid, the software performs an error reset and discards the bad frame by resetting the character counter to zero. If the frame is valid, the *MDATind* (listed as *vpmidi* in the code; see Listing Two, page 93) routine is called and the new buffer pointer is put on the Retix queue for further processing by the portable software. After the buffer pointer is added to the incoming data queue, we get the address of the next available buffer from the *Baddr* array and set the proper pointers so that it is now available for new incoming data. The data receiver routine has a higher priority than the transmitter routine; therefore, at the end of the receiver routine we check to see if a transmit interrupt is pending. If so, that character is processed before leaving the interrupt routine.

Transmitter

On the transmit side, when a buffer is first sent to the *MDATreq* routine (see Listing Two), we check to see if another frame is currently being transmitted. If one is, we return to the calling routine and try again later. If no frame is being transmitted, a test is made to see if the SCC transmit buffer is empty. If it is, the pointers and counters are set and the first character is sent. At that point, we confirm the buffer and start the end of frame processor. The transmit buffer is again checked to see if it is empty and when it is we send the second character. The interrupt driver will continue to send the remaining data until the last byte of the frame is delivered along with the frame's checksum data. The reason two characters are sent instead of one is because the amount of processing that must be done after the first character is sent does not allow us to return from this routine before the transmit buffer of the 82530 chip empties. The transmit underrun condition will result in a false end-of-frame and checksum generation. We essentially "prime" the SCC transmit shift register with enough data to allow the driver to stay ahead of the SCC transmitter during the first few bytes of frame transmission.

System Polling Routine

As we discussed earlier, the system employs a combination polled and interrupt-driven environment. The portable software is executed out of a polling loop and is driven by the data from the SCC. After the hardware and buffer initialization have been completed, the *polldat( )* routine (see Listing Three, page 92) manages the preallocated, receiver buffer array. As a buffer is used by the receiver routine, the pointer in the array is set to NULL and the buffer array counter, *rbufcnt*, is decremented by one. If the buffer counter ever goes below four, then the test in *poldat( )* will be true. Each NULL pointer in the

(continued from page 32)
Baddr array is then given a new buffer address. This allows the receiver routine to always have a fresh supply of buffer pointers. Buffer management is depicted in Figure 2.

Interoperability Results

Our test bed has been used to prototype several query/response applications that run over the transport layer and, in another environment, the network layer. The applications include weather database transfer and maintenance messaging. Furthermore, we have been able to successfully interoperate with a VAX utilizing DEC OSI software as well as a Tandem computer implementing Tandem X.25. Successful tests

The OSI Reference Model

A system can be "closed," whereby components are not interchangable with another supplier's equipment (or alternate technology) without affecting other parts of the system; or, using well-defined, publicly available interface standards that treat components as black-boxes, the system can be "open." The advantage of an open system is its access to multivendor systems and growth without requiring totally new systems.

The International Organization for Standardization's (ISO) Open Systems Interconnection (OSI) Reference Model is an open system approach for communications interfaces. As illustrated in Figure 3, standardization reduces the number of unique interfaces across a network. In the example, four closed systems require 12 unique interfaces to communicate with each other. With a standard set of rules and procedures, like OSI, only four unique OSI-to-native environment interfaces are required. This approach reduces the cost of system development, procurement, support, and maintenance by eliminating the need to develop, debug, and maintain custom interfaces.

The OSI reference model is a framework divided into seven data communications function layers: physical, data link, network, transport, session, presentation, and application. The OSI architecture presents few new ideas to data communications: It is just a way of dividing the functions required for data communications into manageable and understandable subsets. Communications systems designed without using the OSI Reference Model contain many of the same functions as OSI-based systems; these functions, however, are generally lumped together into several undefined layers. OSI provides *primitives*, a minimum, well-defined flow of information between the layers. The implementation of these primitives is unique to a system, but the information passed with these primitives is standard across all systems. This modular approach to communications eases software maintenance since it lends itself to the development of modular, maintainable code.

Within each function layer, several international standard protocols exist

have also been conducted over a small packet switch with two PCs functioning as the end-systems. The PC-based system has to provide the flexibility to interoperate in a heterogeneous hardware and software environment.

Time, Skill, and Resource Requirements

Development of the four-layer system and application, including learning curve, required approximately 15 staff months. The staff working on a project of this nature should possess layered protocol, real-time software development, and testing knowledge. This type of project requires the same hardware and software tools as any real-time project. At a minimum, a logic analyzer, oscilloscope, in-circuit emulator, and

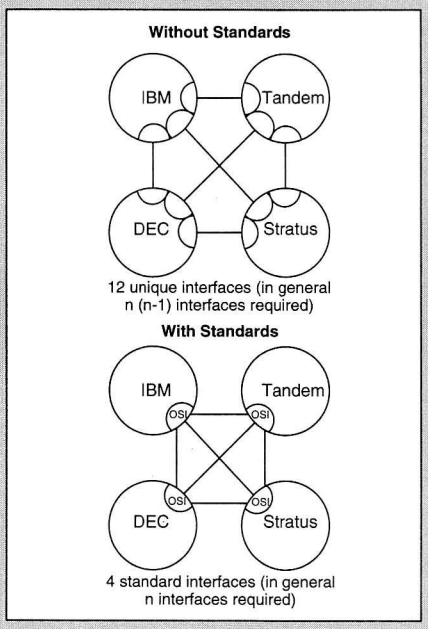


Figure 3: *Benefits of standards*

for implementation within the realm of the OSI Reference Model. GOSIP (the U.S. Government OSI Profile, see *Federal Information Processing Standard 146*) further defines the reference model into sets of protocols (or profiles) that will be used for government data communications systems. The OSI suite can be viewed as providing *transport* and *application* services. The transport services (physical, data link, network, transport) provide communications-related functions (such as network routing and guaranteed delivery of data). The application services (session, presentation, and application) provide a toolkit of features for the applications programmer, so you do not need to know the details of data communications to write an application.

For example, if you need to write a word processor that includes file transfer capabilities, you would design the word processor to use the the File, Transfer, Access, and Management application-layer protocol (see *ISO 8571*). Using the standard application-layer protocol, the basics of file transfer, such as the use of synchronization points for roll-back and management of the virtual filestore, would be handled in the application-layer protocol, not the word processor. This layering of functions is the basis for the OSI Reference Model. If you need to design a special application not covered by the standard application-layer protocols, you could use the other application services features (like data translation and remote operation support) without having to redevelop these functions.

— K.C., M.T.

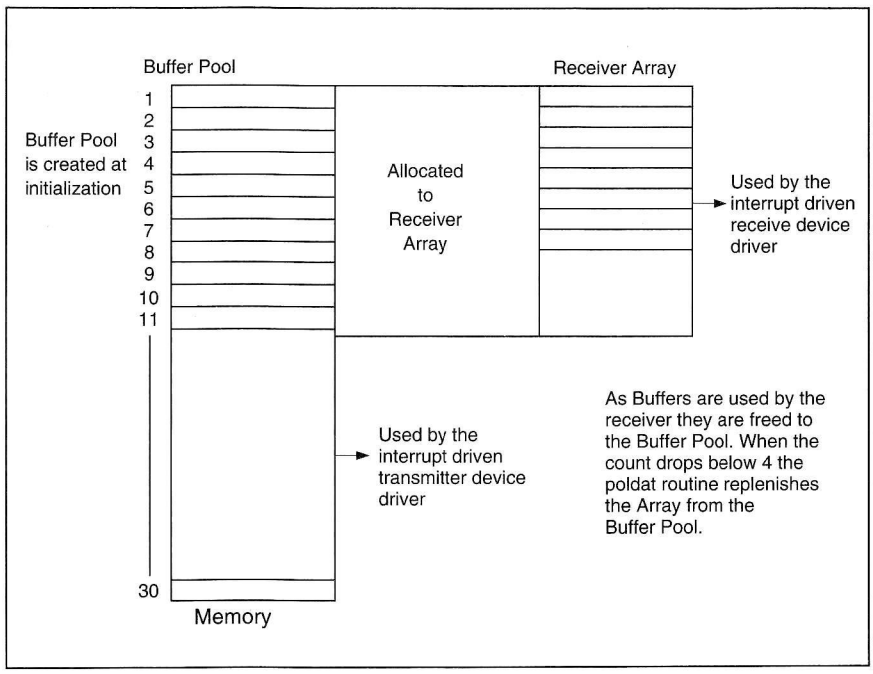


Figure 2: *Buffer management*

protocol analyzer with simulation capability should be available. Code-level debuggers (such as Microsoft CodeView) are useful in debugging the application code; however, they do interfere with the individual protocol timers of each layer and the timing requirements of moving data up and down the communications stack. Software in-circuit emulators that utilize the protected mode of the 80386 and allow you to run the test application as a virtual machine (for example, Nu-Mega SoftIce) are very useful. If an 80386-based machine is available, this is a cost-effective substitute for a hardware in-circuit emulator.

Recommended Development Process

Development of the test bed has shown that the device driver is the most critical part of the system design. The device driver should be designed, tested, and tuned before the upper layers are added. Most likely, vendor-unique management interfaces will need to be implemented with the device driver. The device driver can be written in the native language of the portable code to ease the interface to the link layer, and the assembler output of your compiler should be analyzed to determine if further optimization is necessary. In our case, Microsoft C 5.1 (compiling for execution speed optimization) produced tight assembler code that required little hand-tuning outside of that necessary to meet the 82530 cycle and reset times. Once the device driver and management routines have been developed, work can then progress on porting the upper layers. Finally, the application can be written, but the timing, polling, and interrupt activity must be considered at every phase of the system debugging.

Conclusions

The field of OSI software is still young, and technical risks will continue to decrease as the field matures. The test bed development effort has shown that at least the lower four layers can be implemented today using commercially available portable source code. Our work has confirmed that it is much easier to implement portable source code than to redevelop it for a particular system. However, the use of portable source code is not just a "compile-and-go" situation, since it is essentially a large real-time system driven by the data of another OSI end-system or relay-system.

Acknowledgments

The authors wish to thank Michael McGurrin, group leader, and Dr. Paul T. R. Wang, lead engineer, both of The MITRE Corporation, for their efforts in developing the test bed.

DDJ

(Listings begin on page 92.)

Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

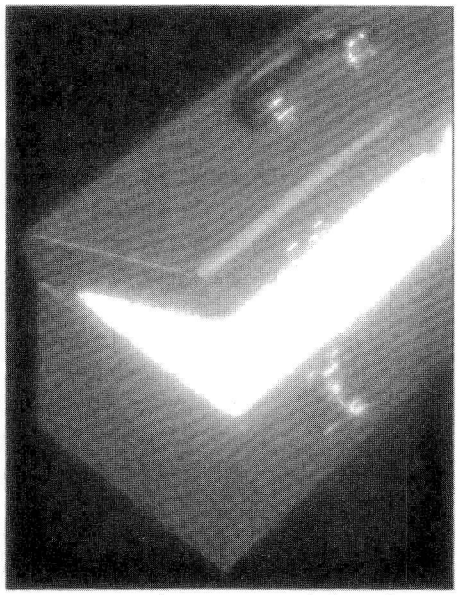
The Macintosh Communications Toolbox

Writing comm applications that are terminal, file transfer, and connection independent

Don Gaspar

Along with standardized communications functions, Apple's Macintosh Communications Toolbox provides an entire spectrum of dynamic tools to aid you in constructing dynamic communications applications. Programs that make use of the Comm Toolbox become completely terminal, file transfer, and connection independent. If you have to add any new functions *and if your program is written properly*, your communications program will be extended with the new functionality. The Comm Toolbox does this via tools kept in the System Folder that offer independence for the communication applications.

Before putting the Comm Toolbox to work for you, become familiar with the Connection Manager, Terminal Manager, and File Transfer Manager. (A fourth key manager — the Communications Resource Manager — will not be discussed in the context of this article.) As Figure 1 illustrates, these managers interact with specific tools that work directly with the ROM OS. The three managers expand the functionality of the Macintosh Toolbox by offering functions for any type of application that involves communications. The managers are simple to use, and their functions resemble those of the other toolbox managers you've already been using for years. If you've used the Memory Manager, TextEdit, QuickDraw, the Window Manager, and some of the others, you're ready to go.



Don is a software engineer at Apple Computer. He can be reached at Apple Computer, 20525 Mariani Ave., MS-35Q, Cupertino, CA 95014. Don was working for Dialog Information Services at the time he wrote this article.

The Connection Manager

The Connection Manager is a medium through which you tell the Comm Toolbox what type of physical connection is between your Mac and the device it's talking to. This "other" device might be a modem, a direct serial connection, ADSP, perhaps a software MNP level 5 or X.25 tool, or maybe something that hasn't even been thought of yet. No problem. Just write your own connection tool and anyone who writes a program using the Comm Toolbox will be able to use your new hardware connection as well.

For instance, once the user selects the type of connection being used (or sets a default), special preferences can then be made for that particular connection. If using a modem tool, the user should be able to tell the program what type of modem is being used, the phone number to dial, baud rate, stop bits, parity, and so on.

The function that enables this parameter setting is *CMChoose( )*. When called, the Comm Toolbox displays a dialog box (see Figure 2) that allows the user to change settings. This is the standard dialog box offered through the Comm Toolbox; you don't have to use it if you don't want to. You can override anything you don't like, then design and use your own. For consistency across communications programs, however, it's recommended you stick with the standard so that someone unfamiliar with your program will understand how to use it.

The Terminal Manager

The Terminal Manager tells the Comm Toolbox what type of emulation is taking place: TTY, VT-100, VT-320, or even IBM 3278. With the Comm Toolbox, you just add a terminal tool with the emulation you want in the system folder; your program then becomes terminal

(continued from page 38)
independent. Apple has implemented several of these emulations, including VT-320. You can use the ones that already exist, license more from a third party, or write your own.

The Terminal Manager's counterpart to the Connection Manager's *CMChoose( )* is *TMChoose( )*; the dialog changes according to what type of terminal is selected and allows the user to change their preference for the particular terminal they are using (the tool controls this).

The File Transfer Manager

The File Transfer Manager allows you to implement standard file transfer (Kermit, XModem, ZModem, text transfer, and so on) functions with minimal effort, for file transfer independence. Suppose, for instance, that next year a new file transfer scheme is implemented on CompuServe. What would happen to your program? Would you have to send users new disks with the new file transfer added to it? Not at all. Users simply add the new file transfer tool to their System Folder, and because your program is file transfer independent, it can automatically use the new tool without any modifications. In fact, the current file transfer tool can be used to get the new one.

A Sample Comm Toolbox Application

In this section, I'll briefly describe a communications application I call "CheapCom" (see Listing One, page 94). This program, written using MPW C 3.1, illustrates how the Comm Toolbox managers and other functions are implemented to easily build a full-featured communications program that supports connection, terminal emulation, and file transfer. Because of space constraints, I haven't included any of the standard Macintosh user interface features in Listing One; the code presented is the communications core. However, the complete system (including resource files), as well as just the core code, is available online (see Source Code Availability, page 3, for details). To compile and use the program, you'll need the Comm Toolbox installed on your Macintosh, MPW with the C compiler Version 3.1 or later, and the Comm Toolbox Interface. (Contact APDA, Apple Computer Inc., 20525 Mariani Ave., Cupertino, CA 95014, for Comm Toolbox specifics.)

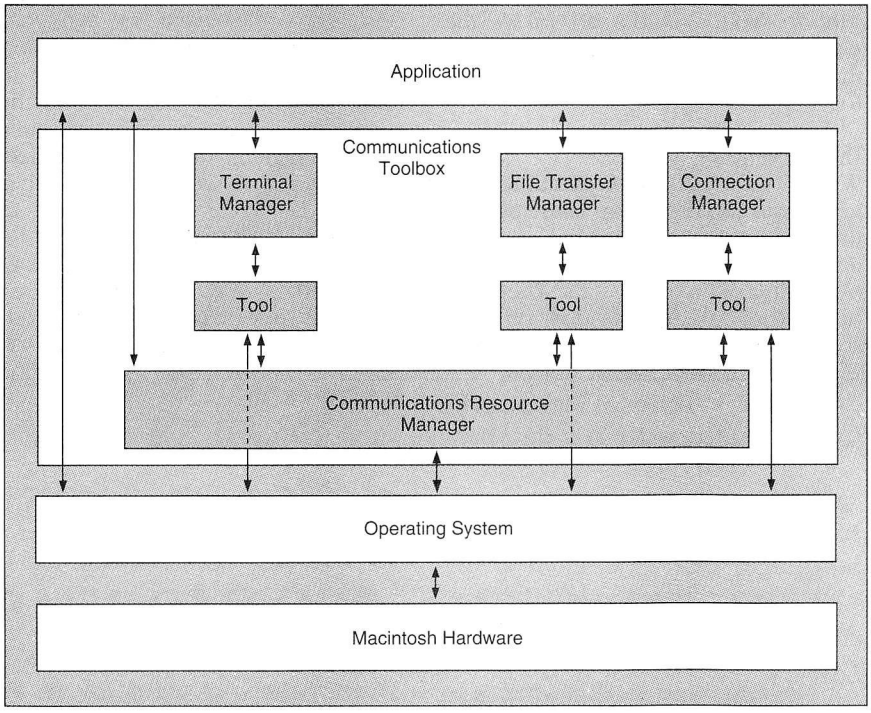


Figure 1: *How the Communication Managers intereact*

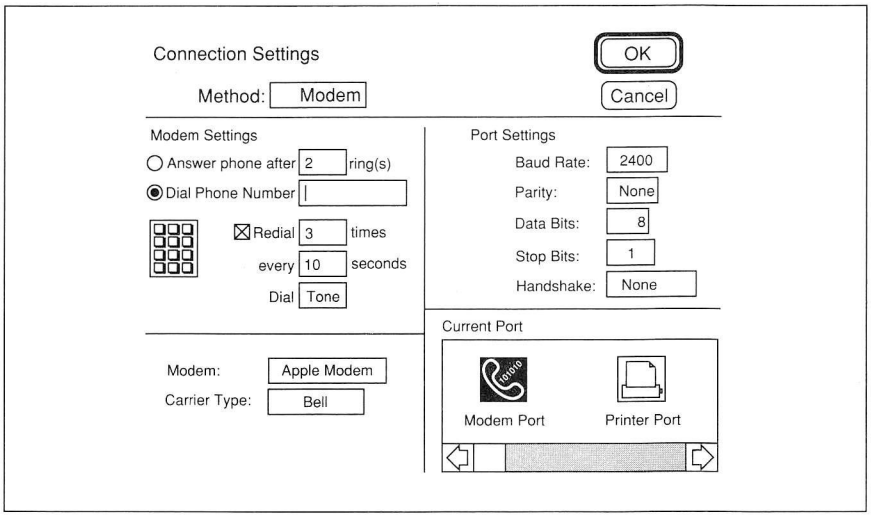


Figure 2: *The Connections Manager dialog box*

To start off, you have to initialize the communication managers just as you do the Window Manager, TextEdit, QuickDraw, and the others. A typical initialization function might resemble that in Figure 3. For compatibility reasons, always initialize the Comm Toolbox in the order shown in Figure 3. You should also always check for existence of the Comm Toolbox before initializing it, otherwise the error codes returned will be meaningless. (If the user is not using System 7.0, then the Comm Toolbox must be installed via a special INIT, available from APDA.) If the Comm Toolbox isn't there when you start up, your program will either have to disable its communications functionality or it will have to just quit (after notifying the user, of course).

Next, you should check for tools. If no tools are available, the value "xxNoTools" will be returned; you might want to quit your program in that case. For example, if no connection tools were available, it would be physically impossible to establish a hardware connection. So you must alert the user and quit the application. However, if no file transfer tools were available, rather than quit the program you might simply alert the user that the program couldn't find any and disable the user's ability to select functions that deal with file transfer.

Then, you need to make a new terminal record, connection record, and file transfer record. Code like that shown in Figure 4 accomplishes this. The calls *TMNew( )*, *CMNew( )*, and *FTNew( )* set up the appropriate records for your application to start with communications; these calls resemble *TENew( )* from TextEdit, and will be familiar to Mac programmers. You also have to set

```
InitGraf((Ptr) &qd.thePort);                    // QuickDraw
InitFonts();                                    // Font Manager
InitWindows();                                          // Window Manager
InitMenus();                                    // Menu Manager
TEInit();                                       // Text Edit
InitDialogs(nil);                               // Dialog Manager
InitCursor();                                   // starting cursor

err = InitCTBUtilities();   // Comm Toolbox Utilities
err = InitCRM();                                // Comm Toolbox Resource Manager
err = InitTM();                                 // Terminal Manager
if (err == tmNoTools)
            AlertUser("No terminal tools found\0", true);
err = InitCM();                                 // Connection Manager
if(err == cmNoTools)
            AlertUser("No connection tools found\0", true);
err = InitFT();                                 // File Transfer Manager
if(err == ftNoTools)
            AlertUser("No file transfer tools found\0",false);
```

Figure 3: *Initializing the Comm Toolbox*

up a buffer at this point, since the Comm Toolbox must have access to it here.

Up to this point, all we're doing is setting up the Comm Toolbox to our specification. However, there's a lot going on in the background. For example, if we didn't want the Comm Toolbox to automatically handle the terminal emulation window and instead create some iconic wonderland for users when we call *TMNew* with *tmSaveBeforeClear + tmAutoScroll*, we could add *tmInvisible* to tell the Comm Toolbox we're handling our own emulation appearance. Since there's much more you can do, you should read *Inside the Communications Toolbox* (the manual that accompanies the Comm Toolbox) for more information.

The Idle Function

When you write a communications program, you usually set up some type of timer to read from the serial port at a timed interval. This prevents hardware overwrites of incoming data. Then you display the data in a window on your screen by reading periodically, possibly every *n*th time through your event loop. In order to handle file transfers, emulation, and data arriving via your comm buffer, you have to read this data and figure out exactly where it goes; the Comm Toolbox aids you with the functions *TMIdle()*, *CMIdle()*, and *FTIdle()* — these functions assist you in delegating the data to the appropriate managers. The function *DoIdle( )* (see Listing One) takes care of this; please take a look at it, since yours will have to be nearly identical. This function updates the file transfer status; the terminal emulation is handled by the Comm Toolbox.

You still have to provide your own customization and buffer. The Comm Toolbox only buffers the terminal emulation window; when the data goes off the screen, you'll have to save it to your global buffer. (Refer to the function *cacheProc( )* in the source code for an example.) When the data reaches the region that is outside of the terminal emulation region, *cacheProc( )* is called; at this point, you would have to either add the data (one line at a time) to TextEdit, or add it to your own linked-list of text that you maintain yourself. Your own linked-list of data objects is preferable, since TextEdit is slow and limited.

(continued on page 44)

```
gTerm = TMNew(&theRect,&theRect, tmSaveBeforeClear + tmAutoScroll,
                        procID, window, (ProcPtr)TermSendProc,(ProcPtr)cacheProc,nil,
                        /*(ProcPtr)clikLoop*/nil, (ProcPtr)ToolGetConnEnvirons,0,0);
if(gTerm == nil)
          AlertUser("Can't create a terminal tool\0", true);
          HLock((Handle)gTerm);

/* connection tool */
procID = FindToolID(classCM);
if(procID == -1)          // get out of here if no tools!
          AlertUser("No connection tools found/0", true);

sizes[cmDataIn] = kBufferSize;        // just data channel; large incoming buffer
sizes[cmDataOut] = kBufferSize;
sizes[cmCntlIn] = 0;
sizes[cmCntlOut] = 0;
sizes[cmAttnIn] = 0;
sizes[cmAttnOut] = 0;

gConn = CMNew(procID, cmData, sizes, 0,0);
if(gConn == nil)
          AlertUser("Can't create a connection tool/0", true);
HLock((Handle)gConn);

/* allocate space for reads/writes using number returned by connection tool */
gBuffer = NewPtrClear(sizes[cmDataIn]);
if(MemError() != noErr)
          AlertUser("Out of memory\0", true);

/* file transfer tool */
procID = FindToolID(classFT);
if(procID == -1)
          AlertUser("No file transfer tools found\0", false);

/* no read/write proc -- tool has its own */
gFT = FTNew(procID, 0 ,(ProcPtr)FTSendProc, (ProcPtr)FTReceiveProc,
          nil, nil, (ProcPtr)ToolGetConnEnvirons, window, 0L,0L);

if(gFT == nil)
          AlertUser("Can't create a file transfer tool\0", true);
```

Figure 4: *Code to make a new terminal, connection, and file transfer record*

(continued from page 41)

Other Modifications

There are a lot of simple things common sense will tell you need to be added to your program. For example, you need to answer questions like "Where does the Terminal Manager get updates and key events?"

Your standard update loop will have to be able to notify the Terminal Manager that it needs to update its emulation region on the screen. You can do this by verifying the terminal record's integrity (not NIL and not odd) and then calling *TMUpdate( )*. This is pretty simple and allows your program to add communications without deviating significantly from a standard Macintosh program.

Suppose the user is typing on the keyboard. That action (event) will have to be translated if terminal emulation is taking place. You'll have to add a couple of lines which make a call to *TMKey( )* to the function that handles normal key events, so that the Terminal Manager will be able to handle such events.

There are several other similar Terminal Manager functions to look at; again, refer to *Inside the Communications Toolbox*. Functions such as *TMResize( )*, *TMClick( )*, and *TMScroll( )* will be devoted to the user selecting text (or objects) within the terminal emulation region of the screen, scrolling (so that data is not lost), resizing the window (so the emulation region is affected), and so on.

If your communications window (perhaps one of several windows your program is using) is in the background and is then activated by the user clicking on it, you address this activate event, bring your window to the front, and call *SetPort( )* so that the Mac will know which grafPort to draw in. Everything is pretty much the same here, except that you might want to add the calls *TMActivate( )*, *CMActivate( )*, and *FTActivate( )* so that the three managers can activate their functions appropriately.

Cleaning up when your program is complete requires that you cancel the functions of the three managers and dispose of your buffer. We can use the Memory Manager function *DisposePtr( )* to get rid of our buffer, but what about the records for the Connection Manager, Terminal Manager, and the File Transfer Manager? Just use the routines the Comm Toolbox provides: *CMDispose( )*, *TMDispose( )*, and *FTDispose( )*. Adding these calls should make everything look like a regular Mac program.

Installation

If you're still working with System 6, you'll have to install the Comm Toolbox (it's not built-in to 6) and create a new folder within the System Folder called "Communications" to put all of your tools in.

If you're distributing an application you built with System 6, you'll also need to offer your customers an installation disk. You can get this, along with several connection, terminal emulation, and file transfer tools from Developer Services at Apple.

Acknowledgments

The author is indebted to Mark Baumwell, James Benninghaus, Veronica Dullaghan, and Alex Kazim, all from Apple. They provided a substantial amount of technical support and help when needed.

DDJ

(Listing begins on page 94.)

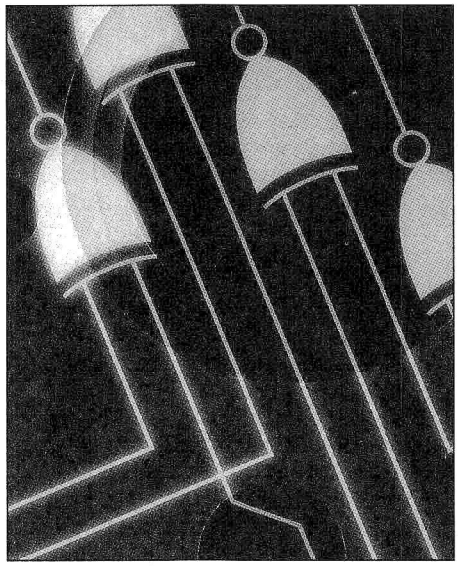
Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

Algebraic Codes for Error Detection and Correction

Controlling data transmission errors

Hsi-Chiu Liu

Electronic data transmission errors are a fact of life and, despite rapid advances in digital communication and computer networks, transmission error control continues to be a major software- and hardware-engineering task. One of the most efficient methods of error detection and correction is *algebraic coding*, which requires only a minimal amount of bit redundancy in forming code words. Using algebraic operations with a matrix and a vector, code words can be easily encoded before transmission and decoded at the receiving end. When compared to other error control schemes, algebraic coding is potentially capable of correcting multi-bit errors with lower-bit redundancy overhead.



Code Word Formation

Algebraic codes are a series of code words, each of which is formed by attaching a number of check bits to a message word. The purpose of the attached check bits is to help detect and correct transmission errors. To correct multi-bit errors, more check bits are needed. For an algebraic coding, the inequality $n + r + 1 <= 2^r$ is used to determine the minimum number of check bits needed for correcting single-bit errors (n is the number of bits of a message word and r the number of check bits needed). For example, a 4-bit message word will need at least three check-bits in order to be able to correct single-bit errors.

Suppose that a message word is n-bits long and it takes r check bits to form a code word. For each correctly formed code word, there may be (at most) $n+r$ code words, each of which is in a 1-bit error. Each of these erroneous code words is formed by inverting any one of the $n+r$ bits of the correctly formed code word. For a transmission that causes only single-bit errors, there will be $n+r+1$ possible code words that may be received in transmitting every code word. Because there are 2^n message words, a maximum of $2^n (n+r+1)$ code words will be involved in the transmission. For a code word that is composed of $n+r$ bits, there will be a total of $2^{(n+r)}$ possible code words, even though it is not possible for some of them to be present in the transmission because of the single-bit error transmission. Based on this reasoning, we have $2^n (n+r+1) <= 2^{(n+r)}$. This inequality is easily reduced to $n + r + 1 <= 2^r$.

Actually, the probability of a multi-bit transmission error is much lower than a single-bit transmission error. For example, if the bit error rate is 10^{-3}, then the probability of a double-bit in error will be 10^{-6}, if the bit error is independent. (That's why I'm only considering single-bit errors in this article.) To generalize this type of coding, we suppose that the message words consist of n bits each. To form a code word, a number of r check bits will be appended to the message word: $m_1 m_2 m_3 \ldots m_n c_1 c_2 c_3 \ldots c_r$. Note that there are only 2^n valid code words out of 2^{n+r} possible ones.

Code Word Encoding

To generate the r check bits algebraically, we have to first predefine a matrix called "H" of the dimension r x $(n+r)$. Consider each code word to be generated as a vector T which consists of message bits followed by check bits. If even parity is adopted for the computing system, appropriate values can

Hsi-Chiu Liu is an associate professor of computer science at California State Polytechnic University, Pomona, Pomona, CA 91768.

(continued from page 46)
be assigned to each of the check bits from the matrix equation H T = 0. See Example 1 for definitions of H and T. Note that the righthand portion of H is an identity matrix. The entries in the lefthand portion of H must be either 0 or 1. These values must be predefined under the following two conditions: No column consists of all 0s; No identical entries are assigned to any two columns. The following problem illustrates how to assign values to the check bits.

$$H = \begin{bmatrix} h_{11} & h_{12} & \cdots & h_{1n} & 1000\cdots0 \\ h_{21} & h_{22} & \cdots & h_{2n} & 0100\cdots0 \\ & & \vdots & & \\ h_{r1} & h_{r2} & \cdots & h_{rn} & 0000\cdots1 \end{bmatrix}$$

$$T = \begin{bmatrix} m_1 & m_2 & \cdots & m_n & c_1 c_2 \cdots cr \end{bmatrix}$$

Example 1: *Definition of H and T*

(a)

$$H = \begin{bmatrix} 1101100 \\ 1011010 \\ 0111001 \end{bmatrix}$$

(b)

$$T = \begin{bmatrix} 1101c_1c_2c_3 \end{bmatrix}$$

(c)

$$1+1+0+1+c_1+0+0=0$$

$$1+0+0+1+0+c_2+0=0$$

$$0+1+0+1+0+0+c_3=0$$

(d)

$$c_1 = 1$$

$$c_2 = 0$$

$$c_3 = 0$$

Example 2: *Solving the message word 1101*

Problem: Given the message word 1101 ($n = 4$), then the number of check bits is set to 3 ($r = 3$). If the predefined matrix H is as shown in Example 2(a), find the values of the check bits for the formation of a code word with the given message word.

Solution: First, a vector for the message word 1101 is formed. See Example 2(b). Secondly, assume that even parity is adopted. Then, perform the matrix multiplication: H T=0. Three equations will be generated from this matrix multiplication; see Example 2(c). Using modulo-2 addition, the three equations will be reduced to Example 2(d).

Therefore, the valid code word will be 1101100. Following this procedure, all of the valid code words formed with all of the 4-bit message words can be generated; see Example 3.

Code Word Decoding

Let's next consider how to detect and correct transmission errors. Assume that the received code word is a vector R. We also assume that an error vector E is of the same dimension as that of the code word or vector R. The 1s in E

(continued from page 48)
represent the error positions in the code word. We then have R = T+E. Now, use the following to perform a matrix multiplication: H R = H(T+E) = H T + H E = 0 + H E = H E.

Therefore, if this multiplication results in product zero (i.e., H E = 0), we then conclude that E consists of all 0s.

Message word	Code word
0000	0000000
0001	0001111
0010	0010011
0011	0011100
0100	0100101
0101	0101010
0110	0110110
0111	0111001
1000	1000110
1001	1001001
1010	1010101
1011	1011010
1100	1100011
1101	1101100
1110	1110000
1111	1111111

Example 3: *Valid code words formed from 4-bit message words*

(a)

$$\begin{bmatrix}1101100\\1011010\\0111001\end{bmatrix}\begin{bmatrix}1\\1\\0\\0\\1\\0\\0\end{bmatrix} = \begin{matrix}1\\1\\1\end{matrix}$$

(b)

$$S = \begin{bmatrix}1\\0\\0\end{bmatrix}$$

Example 4: *Solving the code word T = 1101100 and the received code word R = 1100100*

This means that there is no transmission error detected, and no transmission error has occurred. Otherwise, at least one error was made in transmission. Of course, this elaborate system will do more than just error detection.

The value of the algebraic coding arises in multiple-error detection and correction. The product HE is defined as S, the syndrome. The dictionary defines syndrome as "a number of symptoms occurring together and characterizing a specific disease or condition." In fact, the error syndrome characterizes the specific bit error.

Let's first assume that a single-bit error occurs. E would then be a vector composed of a single one, and the remaining positions would be zeros. If we take the product of this with H to form the syndrome, the result is a vector that is identical to one column of H, that column being the one corresponding to the bit position in error.

To illustrate the single-bit error correction mechanism, consider this problem:

Problem: Given the code word T = 1101100 (before transmission), and the received code word R = 1100100 (after transmission), find the error and correct it.

Solution: First form the product HR; see Example 4(a). Note that the result is equal to the fourth column of H, thus identifying an error as having occurred in the fourth-bit position. This error can be corrected simply by inverting the fourth bit of the received code word.

If an error of more than 1-bit error occurs, the syndrome will be equal to the sum of the corresponding columns of H. If, for example, errors occurred in the third and fourth bits of the example above, the syndrome would be as in Example 4(b).

This would be incorrectly interpreted as a single-bit error in the fifth bit position. Even though in this article we restrict our discussion of algebraic decoders to the single-bit error correction case, note that the syndrome can correct two errors, provided that no two columns sum to either zero or to another column or to the sum of two other columns.

Implementation

Algebraic codes are not difficult to implement. Calculation of the various vec-

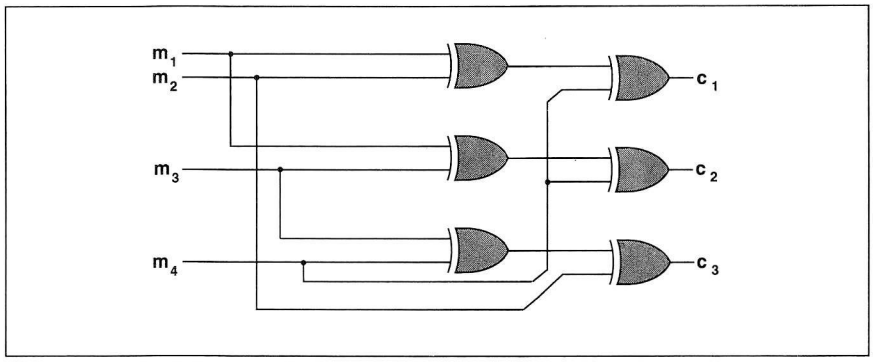


Figure 1: *Check bit encoding*

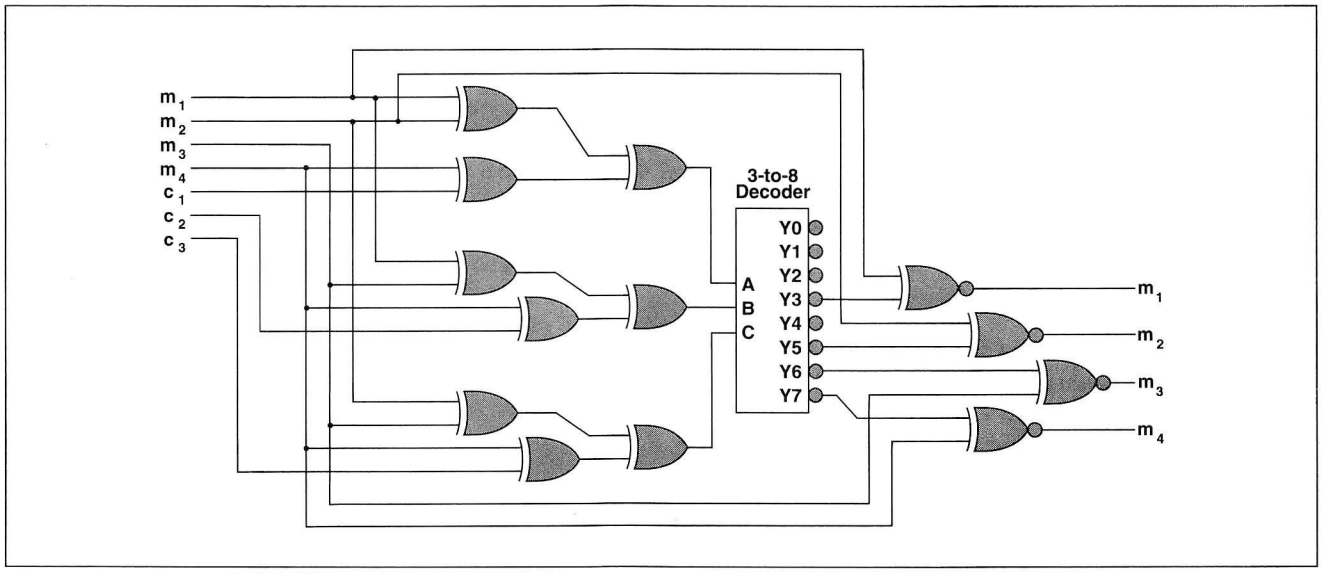


Figure 2: *Code word decoding*

tor products is equivalent to summing combinations of bit positions. This is accomplished by entering the codes into shift registers, tapping off at the appropriate positions, and feeding these outputs into a summation device. Modulo-2 addition of two inputs is performed in the XOR logic block.

One simple implementation of the algebraic coding and decoding system for 4-bit message words is shown in Figures 1 and 2. In Figure 1, the 4-bit message word to be transmitted is input from the left. The correct values of the check bits for this message word are output from the right. So, a code word can be formed with the message word and the check bits. In Figure 2, the received code word is input from the left. The correct message word is output from the right. Note that the implementation setup in Figure 2 is capable of handling both detecting and correcting single-bit errors. Another possible implementation using ROM is shown in Figure 3. Note that in part (a), each message word to be transmitted is used as an address pointing to the ROM location where the needed check bits are stored. Similarly, in part (b), the received code word is also used as address pointing to the ROM location where the correct message word is stored.

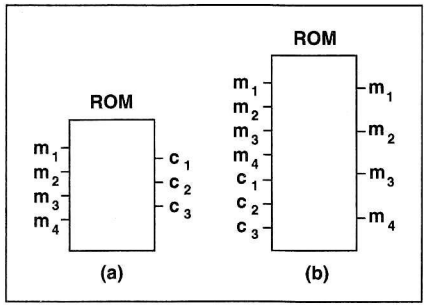


Figure 3: *ROM implementation: (a) Check bit encoding; (b) Code word decoding*

References

Roden, Martin S. *Digital Communication Systems Design*, Englewood Cliffs, N.J.: Prentice Hall, 1988.

Tanenbaum, Andrew S. *Computer Networks, Second Edition*, Englewood Cliffs, N.J.: Prentice Hall, 1988.

White, Ben. "Hamming-Code Decoding," *Dr. Dobb's Journal*, #156, October 1989.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 4.**

Supercharging Sequential Searches

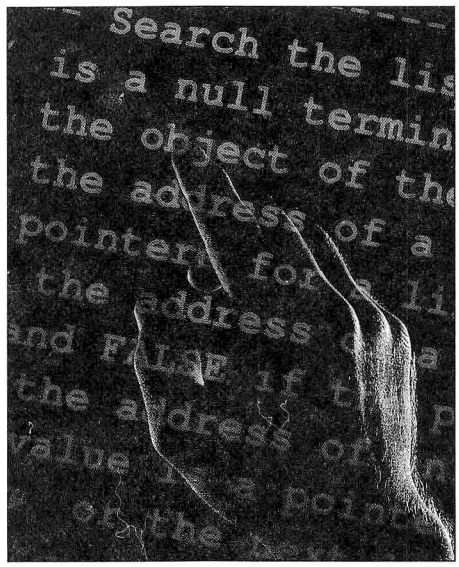
Speed plus compression equals faster searches

Walter Williams

We all know a sequential search is slow. Search time increases linearly with the size of the list; and as a list grows beyond a few items, the search time quickly becomes unbearable. Nevertheless, because it is easy to code, works on just about any list, and provides acceptable speed for short lists, the sequential search remains one of the most commonly used search algorithms.

Consequently, there is much to be gained by speeding up the sequential search algorithm, while maintaining its inherent generality and simplicity. This article describes a simple algorithm that can often speed up a sequential search by a factor of two or more. And there's a special bonus — a list can also be compressed, often to half its original size. The improvement results from a better method of comparing each key. The number of key comparisons is the same as with any sequential search, but the time spent comparing each key is dramatically reduced.

As with any sorted sequential list, the number of keys compared will be, on average, half the number of keys in the list. The number of key comparisons, however, is not the whole story. Each key comparison is itself a sequential search (a search for a non-matching character), so the number of character comparisons (I'll assume the keys are character strings) is N / 2 * K / 2 (or NK / 4) where N is the number of items in the list and K is the key length.



A sorted list, however, presents us with an interesting opportunity. The opportunity arises from the fact that the sort brings similar keys together. Often, the first few characters of one key duplicate the first few characters of the preceding key. As a consequence, a typical sequential search spends much of its time comparing the same leading characters over and over. If those redundant characters are skipped the search will be faster. The approach described here, called the suffix list, speeds up the search by eliminating those needless comparisons. Here's how it works.

The list is kept in ascending order and each key is divided into two parts, a prefix and a suffix. The prefix is the portion of the key which matches the previous key. The suffix is the remainder of the key, beginning with the first character that differs from the previous key. The suffix list stores an integer that represents the prefix length along with each key.

Unlike a simple list, in which the complete key for each item is used during the search, a suffix list uses only the suffix in key comparisons. The prefix itself is ignored, because its length — the number of characters that match the previous key — provides all the information needed for the search.

The data can be stored as a linked list, as an array of fixed length items, or as a series of variable length items concatenated one right after the other in contiguous memory. Listing One, page 100, shows code which uses a linked list. It doesn't matter which method you use; the basic principles are the same, although the methods for traversing the list differ.

A linked list has the advantage of making insertion and deletion of items easier, but storing the list in contiguous memory uses less space.

The *seq_cell_S* structure in Figure 1(a) illustrates a typical structure for building a linked list. The *sfx_cell_S* structure in Figure 1(b), on the other hand, illustrates a structure for building a list to be used with a suffix search. The

Walter is an analyst at Phoenix Mutual Life. He can be reached at 5 Burns Ave., Enfield, CT 06082 or 203-745-9159.

(continued from page 54)
only difference is the addition of the element *cell.pfxcnt*, which stores the prefix length.

cell.key is the first byte of a null terminated string that contains the key. Note that the *cell.key* is not a pointer to a string, but is the actual location of the beginning of the string. The *cell.prev* and *cell.next* elements are pointers to the previous and following cells in the list, respectively.

Figure 2 shows a list of city names, the prefix counts, and prefix and suffix values. In *seq_cell_S*, the prefix and suffix are not stored separately. The full key is stored in *cell.key*, as normal, but is now supplemented by the prefix length, which is kept in *cell.pfxlen*.

Searching

Like any sequential search, a pattern key is compared to each successive key in the list. The search starts at the beginning of the list and continues until a matching item is found or until an item greater than the pattern is found. An example of code that performs this task is contained in the *Search( )* function in Listing One.

Unlike a standard sequential search, a suffix list search does not examine the actual key value for every item in the list. Instead, the prefix length for each item is compared to a running count of the number of characters matched so far in the pattern. When the search begins, the match count is zero; nothing has been matched. The search progresses, and the match count increases as each character of the pattern is matched until, when the item is found, the match count is equal to the pattern length.

Only when the match count is equal to the item's prefix count does the actual key value come into play.

If the prefix length is greater than the number of characters matched in the pattern, the search skips directly to the next item in the list. Why? Observe that the next character to compare is part of the prefix for the current key; it is, by definition, the same as the character in the same position in the previous key. It is also the first character in the last key that did not match the pattern. Obviously, if the character did not match in the last key, it will not match in this one. So the search can safely jump to

(a)

```
struct seq_cell_S {
                    struct seq_cell_S *next;    /* Next node */
                    struct seq_cell_S *prev;    /* Previous node */
                    char key[1];                /* Key value */
                    } cell;
```

(b)

```
struct sfx_cell_S {
                    struct seq_cell_S *next;    /* Next node */
                    struct seq_cell_S *prev;    /* Previous node */
                    unsigned char pfxcnt;       /* Prefix Length */
                    char key[1];                /* Key value */
                    } cell;
```

Figure 1: *The* seq_cell_S *structure in (a) illustrates a typical structure for building a linked list. The* sfx_cell_S *structure in (b), on the other hand, illustrates a structure for building a list to be used with a suffix search. The only difference is the addition of the element* cell.pfxcnt, *which stores the prefix length .*

Standard	Pfxlen	Prefix	Suffix
Acampo	0		Acampo
Acton	2	Ac	ampo
Adelanto	1	A	delanto
Adin	2	Ad	in
Agoura Hills	1	A	goura Hills
Agoura Hills	12	Agoura Hills	
Aguanga	2	Ag	uanga
Ahwahnee	1	A	hwahnee
Alameda	1	A	lameda
Alamo	4	Alam	o

Figure 2: *A list of city names, the prefix counts, and prefix and suffix values*

(continued from page 56)
the next item in the list.

If the prefix length is less than the match count, the search ends in failure — the pattern is not in the list. This happens only when a character position that has already been successfully matched contains a new and different character. The list is in ascending order,

A search of the 250 city names was about twice as fast as a standard sequential search

so that new character would have to be greater than the one already matched in that position. Therefore, the pattern key would have to have come before the current item if it were in the list.

If the prefix length does, in fact, equal the match count, the suffix must be compared to the pattern. The comparison proceeds character by character, beginning with the first character of the suffix and with the first unmatched character in the pattern. The match count is incremented for each matching character. If it turns out that the pattern matches the suffix exactly, the item has been found, and the search is over. If the pattern is greater than the item, the search continues. If the pattern is less than the item, the item is not on the list, and the search fails.

Consider the following example, which searches for *Adept* in the list in Figure 2. When the search begins, the match count is zero. The prefix count of the first item is, of course, also zero. So *Adept* is compared to *Acampo*. Only the first character matches, so the match count becomes 1, and the search continues. The prefix count for the next item in the list, *Acton*, is 2, which is greater than the match count, so it is skipped. The prefix count of the third item, *Adelanto*, is the same as the match count, so the pattern and suffix are compared. The next two characters of the pattern, *de*, match the corresponding characters in the key, so the match count is advanced by 2, to become 3. The next character, *l*, is less than *p*, so the search continues. The fourth item, *Adin*, has a prefix count of 2, which is less than the match count. The search is over because the item is not in the list.

From the previous example it's clear that this algorithm makes relatively few character comparisons. The average number of comparisons will be between N / 2 + K and N + K. (Where N is the number of records, K is the average key length, and the comparison of the prefix count to the match count is a single comparison.) This is generally far less than the average of KN / 4 character comparisons which a standard search would make.

Insertion

Before an item can be inserted into the list, the appropriate location for the item has to be found. After all, the list has to remain correctly sorted after the new item is added. The first step in adding an item, therefore, is to search for the item by using the algorithm described above. In the case of duplicate items, the search must continue to the last matching item.

After the position for the new item is found, the key has to be separated into prefix and suffix. That's easy, because the prefix was already identified during the search. *Search( )* saves the match count from the item just before the position at which the new key is to be inserted. That match count is, by definition, the prefix count for the new item. We just allocated enough memory to hold the cell structure, and then insert the new item, including the prefix length and key value, into the list. The whole process is not much different from that of any other linked list insertion.

But there is a wrinkle. Remember that each item's prefix length depends on the previous item. It's possible that after insertion there will be greater similarity between the new key and the next one. If that's the case, the prefix length of the next item in the list will change. Because the list is sorted, the prefix length will increase if it changes.

The prefix lengths tell us all we need to know to adjust the next item. The prefix length of the new item will never be less than the existing prefix length of the next item — if it were, the list wouldn't be properly sorted. If the prefix length of the new item is greater than that of the next item, the prefix length of the next item will not change. Only if the prefix lengths of the two items are the same will the prefix length of the next item change. In that case, the two suffixes must be compared and the prefix length adjusted accordingly.

Let's insert *Adept* into the sample list. The first step is to search the list, just as we did before. That search ended just after *Adelanto* with a match count of 3. So the new node is inserted into

the list between *Adelanto* and *Adin* with a prefix length of 3. The prefix length for *Adin* is not the same as the prefix length for the new entry, so the prefix length for *Adin* does not change.

It was somewhat surprising, however, to find that the compression improves the speed of the search

Deletion

Deletion of an item from the list is similar to insertion. Again, the first step is to search through the list until the desired item is found; the second step is to remove it.

The actual removal of the item is no different than removal of an item from any other list — memory allocated from the heap must be freed, pointers updated, and so on. But just as with insertion, the prefix length of the next item following the deleted item may change. This time it will get smaller, never larger.

The adjustment of the prefix length depends, not surprisingly, on the prefix length of the item being deleted and the prefix length of the item following it. The new prefix length for the next item will be the lesser of the two.

Let's delete *Adept*. The search proceeds as before, this time ending successfully with *Adept*. We unlink it from the list, but before releasing the memory we compare the prefix length of the deleted item to that of the item following it. The new prefix length of *Adin* is already less than that of *Adept*, so the prefix length for *Adin* does not change.

Compression

You will have observed that the prefix portion of the key is never used. In fact, as far as the search is concerned, it can be eliminated completely. The prefix will never have to be reconstituted for basic list operations. Even when part of the suffix must be rebuilt on deletion, all of the information needed is contained in the suffix of the key being deleted.

By eliminating the prefix, the list can be stored more compactly. That's an obvious advantage if the list is kept entirely in memory. But it's also an advantage if the list must be retrieved from disk frequently — the more compact the data, the greater its chance of being in cache.

How much space does it save? Tests run on a list of 250 city names and zip codes give some idea of the improvement possible. The original list used 20 bytes for each city name. The average length of the actual names was about 8.2 characters and the average suffix length was about 5.7 characters. The suffix structure includes a single *unsigned char* to store the prefix length, and the variable-length keys require a null terminator, so the net result (including 4 bytes for links) is a savings of 20 – (5.7 + 6) = 8.3 bytes per record. That's a 41 percent savings relative to a fixed length table. (See Table 1.)

Of course the amount of space saved depends upon the data. The greater the similarity between keys, the greater the savings. Best of all, when duplicate keys occur, the suffix is null and none of the key is stored.

Performance

A search of the 250 city names was about twice as fast as a standard sequential search. The improvement agrees with predictions based on the formulas for character comparisons presented earlier.

There are a few other tricks for speeding up a sequential search. These include using the search pattern as an end marker, unrolling the loop, or using a self-organizing list.

The self-organizing list is generally the most effective of the three. When the distribution obeys Zipf's law, it takes NK / log(2)N character comparisons. The self-organizing list is a substantial improvement over a standard sequential search; but it is generally not quite as fast as the suffix search. The trade-off between a self-organizing list and a suffix list will not favor the self-organizing list unless the key length is less than the log (base 2) of the number of records.

Type of List	List Size	Percent Saved
Fixed length records	20.0 bytes	0%
Linked list (full keys)	13.2	33%
Linked suffix list	11.7	41%
Contiguous suffixes	7.7	61%

Table 1: *Typical savings provided by compression technique*

(continued from page 60)

Applications

This algorithm was originally devised to search nodes in a B-tree. In a B-tree each node contains keys that are very similar — sometimes all of the keys are identical — so the suffix search is substantially faster than a standard sequential search. But the real payoff is that by eliminating the prefix many more keys fit in a node — and that reduces the number of disk hits, which are relatively time-consuming. A binary search, which is often used to find a key in a B-tree node, is still faster than the suffix search, but the reduction of the number of disk hits more than makes up for the slower search. (If duplicate keys are permitted in the B-tree, the binary search must be followed by a sequential search anyway.)

There are, of course, other applications where keys with similar prefixes are common: directory lists, compiler symbol tables, and so forth. Similar improvements ought to be possible there, too. It makes sense to compress keys by removing the redundant prefix, and that was the original objective of this method. It was somewhat surprising, however, to find that the compression improves the speed of the search. One would expect the elimination of the prefix portion of the key to make list maintenance more awkward. Instead, the prefix count turns out to be more useful than the actual characters of the prefix.

There are drawbacks to the method. If the prefix is eliminated, the keys have to be reconstructed when needed. The programs are also a bit more complex than a standard sequential search. But for many applications, the advantages far outweigh those drawbacks.

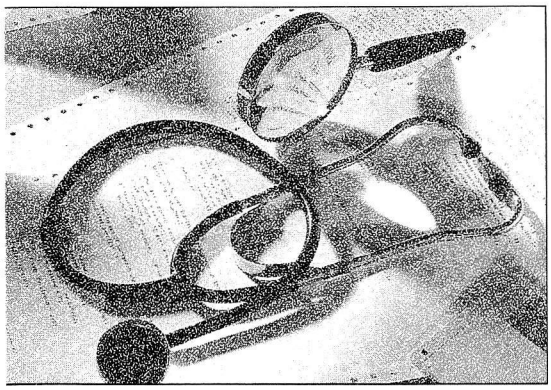
Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listing begins on page 100.)

Vote for your favorite feature/article.
Circle Reader Service **No. 5.**



Examining the Zinc Interface Library

High-powered windowing tools for Turbo C++

Gary Entsminger

Object-oriented programming encourages the reuse of code, primarily through inheritance. A programmer can create classes of objects and then derive new classes from them. But a crucial and (perhaps surprisingly?) subtle aspect of inheritance is that programmers can derive new classes from abstract classes they didn't write. If these classes are general enough to allow themselves to be extended, they can be distributed in the form of extensible libraries, without including source code. Library users can use these classes without necessarily knowing their implementation details, and derive new classes from them.

A great idea, I believe, but one still in an incipient stage. Hybrid object-oriented languages such as C++ and Turbo Pascal, in particular, are still new enough to lack a good assembly of class libraries. Object-oriented thinking *is* substantially different, and requires fresh perspectives. Translating old C and Pascal code just won't do. So it isn't surprising that we're just beginning to see the first wave of commercial libraries that are truly object-oriented, and not just rehashes of structured ideas. In this article I'll examine one of these, an impressive interface library from Zinc Software.

The Zinc Interface Library (ZIL) is a C++ class library that you can use to construct a windowed graphics or text interface for your applications. The interface handles all input (keyboard, mouse, and other device events), creates and manages windows, and routes event information to the appropriate windows. Your applications "run" in these windows.

Gary is a writer, programmer, and consultant, and can be reached at the Rocky Mountain Biological Lab, Crested Butte, CO 81224 and on CompuServe [71141,3006].

Browsing the Package

ZIL is conceptually smart and, when in graphics mode, is similar in appearance to Microsoft Windows or possibly the Next machine. While the people at Zinc have plainly stated that it is *not* a Windows look-alike, ZIL offers a professional-looking user interface without the overhead of a complete environment such as Windows. In text mode, ZIL mostly follows the SAA/CUA guidelines. ZIL is easy to use (once you learn to think in terms of events and messages), and offers an excellent alternative for C++ developers who want to develop graphics applications outside the Windows (and other GUI) environments.

The Library consists of seven main sections. The first section involves event mapping. In fact, the Zinc Interface Library design is an excellent example of an event-driven architecture. In such an architecture, the flow of a program is determined by an event — a force outside the current process, window, or the computer itself.

Events are essentially anything that happen within the system: keyboard, mouse, and other device input; error messages; messages sent to and from objects and devices, and so on.

Related to event mapping is the event manager, which consists of a control unit for input devices and a storage unit for event information. The other sections of ZIL include a window manager (the control unit for any windows added to the screen display), a screen display, help system, error system, and palette mapping (for customizing displays).

Using ZIL, you develop applications that respond to messages triggered by events. You set the interface up (easily), and then set up your application to run in a ZIL window as a derived window object. Each user-activated event (keystroke, mouse, and so on) is routed to the appropriate window, which decides how it's supposed to handle the event.

Applications "running" under ZIL share a similar, general-purpose interface. You can use the interface without having to rewrite (or even know about) the implementation details of the interface. You can add windows, window objects (buttons, icons, and so on), and pop-up and pull-down menus, leaving the management of these objects to the event and window managers.

Each application that uses ZIL works similarly. An event manager looks for input from the keyboard, mouse, and so on, and routes it via an event queue to the window manager. The window manager then sends a message to the appropriate window (see Figure 1).

Because the event manager knows about various displays (CGA, VGA, EGA, Herc, MDA, 43-line, 50-line, and so on) it can determine the appropriate adaptor and display automatically. Thus, an application can be written generally, without concern for specific hardware. If it uses multiple display classes, any application you write can be abstract regarding its screen display. You

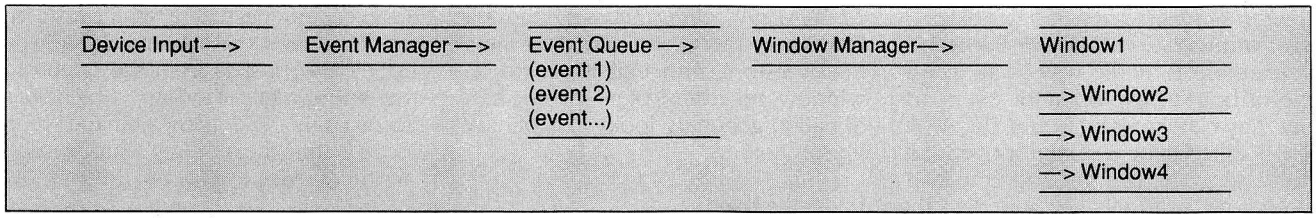


Figure 1: *Design of ZIL*

(continued from page 64)
use one set of source code to generate output for both graphics- and text-based environments.

The screen display controls all low-level screen ouput. The base class for the screen display is *UI_DISPLAY*. It has the ZIL-defined descendants: *UI_DOS_BGI_DISPLAY* and *UI_DOS_TEXT_ DISPLAY*, and support for snow checking on CGA monitors and IBM's Topview (which supports Microsoft Windows and Quarterdeck's Desqview environments). Additionally, you can derive new *DISPLAY* objects and incorporate them in an application as easily as you would ZIL-defined objects.

The ZIL Help System lets you provide context-sensitive help information for end users. You generate a help file using a ZIL utility called GENHELP, and then add the *UI_HELP_SYSTEM* class and its descendants to your application (more on this later).

The ZIL Error System lets you display error information to the end user in a similar manner. Simply add the *UI_ERROR_SYSTEM* and its descendants to your application; when an error occurs, information you specify will be presented to the user in another ZIL window.

The Palette Mapping class allows you to define color combinations for window objects with either global color palette mapping or individual object color palette mapping. Thus, any interface you develop can easily be customized by you or an end user.

Using the Library

To use (and extend) the library, you should create an instance of a window, add library- or user-defined window objects, and add the window to the window manager. Window objects are anything you associate with a window: Objects that compose windows (borders, titles, and scroll bars, for example), objects that specify windows (such as menus and buttons), and objects that are inside windows (such as fields, numbers, strings, and actions).

Incorporating the library into an application means you must construct a screen display and an event manager, add any input devices to the event manager, construct a window manager, and add any windows to the window manager.

To see how this works in practice, the code shown in Example 1 (page 70) initializes the Zinc Interface Library and adds one application window to the window manager. Note that window objects are added to a window dynamically using the overloaded + operator and *new*. The window, including any window object added to it, is similarly added to the window man-

(continued from page 66)
ager with **windowManager + window.*

Example 1, by the way, is an example of how to use operator overloading. Any class derived from *UIW_ WINDOW* inherits the + operator overload. So window objects are added to windows and windows are added to the window manager via the same + operator. The definition is shown in Example 2 (page 70).

You can add functionality to a window by constructing a new window object and adding it to the window, as shown in Example 3 (page 70). Any window derived from the abstract *UI_WINDOW_OBJECT* class has the built-in ability to move, size, minimize, maximize, restore, and exit itself. So any window, regardless of its specific functionality, generally looks and behaves like any other.

Under the Hood

Once we've constructed the event and window managers and added windows (containing applications) to the system, we sit back and let ZIL do the work, essentially through an event loop. In other words, the ZIL event manager gets an event, interprets it and then sends the interpreted information via an event queue to the window manager. The window manager routes the information it gets from the queue to the appropriate window, which acts accordingly. The event manager then loops getting the next keyboard, mouse, or other device event until the window gets an exit request. A simple event loop is shown in Example 4 (page 70).

ZIL offers a professional-looking user interface without the overhead of a complete environment such as Windows

Besides the many basic window objects that describe and manipulate a window, ZIL includes many complex window objects such as prompts, pull-down and pop-up menus, icons, strings, dates, and data fields. To demonstrate window object usefulness, the code segment in Example 5 (page 71) adds database-like input fields to a window.

ZIL's *UIW_NUMBER* class is complete and handles the *char*, *unsigned char*, *short*, *unsigned short*, *int*, *unsigned int*, *long*, *unsigned long*, *float*, and *double* data types. *UIW_NUMBER* also permits various formatting including decimal, scientific, and so on.

A Smart Line Editor

In this section, I'll develop a smart line editor that utilizes many of ZIL's features. The example demonstrates how to derive a new class of objects from existing ZIL classes, construct a data field and add it to the line editor, construct and display event and window managers, add a new instance of a derived class (the line editor) to the window manager, set up and run an event loop, and destruct everything created.

Listing One (page 101) presents the complete source (excluding ZIL header files) for the line editor. The new class derived from ZIL adds file I/O capability to a window. It consists of a constructor that opens and reads a file, a destructor which writes the edited text back to the file, and a buffer for holding characters. Therefore, all the action, including construction of event and window managers, the event loop,

(continued from page 68)
and so on, occurs during construction and destruction phases.

The key to how the line editor works lies in the built-in capabilities of *UIW_STRING*, the ZIL string class. Note, for instance, the **window1->line_buffer* reference in Listing One. The line buffer is updated each time the line displayed on the screen is revised. When we construct the line editor in window 1, we pass the line buffer information contained in a file (in this case, MK.BAT). Then, when we destruct the line editor, the contents of *line_buffer* are written back out to MK.BAT.

Any class derived from *UIW_STRING* and connected to the window manager can automatically detect screen types and devices (such as a mouse), and possesses numerous editing abilities, including cursor and mouse movement, delete word, undo, redo, and so on. For completion's sake, Listing Two (page 102) shows the MAKE file (included with ZIL) used to compile with Turbo C++.

```
// Construct a graphics display if possible.
UI_DISPLAY *display = new UI_DOS_BGI_DISPLAY;
if (!display ->installed)
    // if this system can't handle graphics,
    // delete the graphics
    // display we just created with new.
{
   delete display;
   // then create a text display.
   display = new UI_DOS_TEXT_DISPLAY;
}
  // Construct an event manager.
UI_EVENT_MANAGER *eventManager =
   new UI_EVENT_MANAGER(100, display);
                          // 100 = MaxNoEvents
*eventManager             // add devices by constructing
                          // new instances of device class.
   + new UI_BIOS_KEYBOARD
   + new UI_MS_MOUSE
   + new UI_CURSOR;
  // Construct a window manager.
UI_WINDOW_MANAGER *windowManager =
   new UI_WINDOW_MANAGER(display, eventManager);
  // Construct and add a window to the window manager.
UIW_WINDOW *window1 = new UIW_WINDOW(1,1,50,20,WOF_NO_FLAGS,
                                          WOAF_NO_FLAGS);
*window1 // add window objects by constructing instances
         // of the window object class.
   + new UIW_BORDER
   + new UIW_TITLE("WIN1", WOF_JUSTIFY_CENTER);
```

Example 1: *Initializing the Zinc Interface Library*

```
UI_WINDOW_MANAGER &operator+(void *object)
  { Add((UI_WINDOW_OBJECT *)object); return(*this); }
```

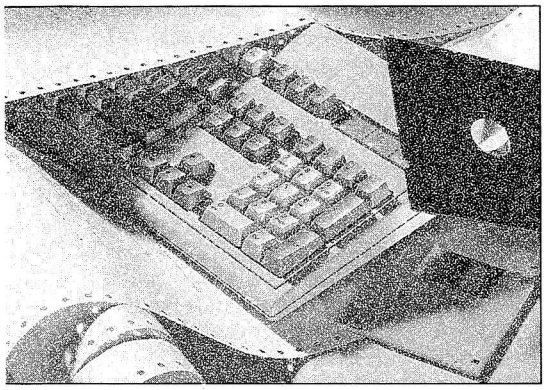
Example 2: *Inheriting an overloaded operator*

```
+ new MY_OBJECT
```

Example 3: *Adding functionality to a window using the + operator*

```
int ccode;
UI_EVENT event; // an event is an instance of class UI_EVENT
do    // loop:  do-while event not equal to ESCAPE
{
     // Get input from a user.
      eventManager.Get(event, Q_NORMAL);

     // If ESC message, exit.
      If (event.type == E_KEY && event.rawCode == ESCAPE)
          event.type = L_EXIT;

     // Send event information to the window manager
      ccode = windowManager.Event(event);
}  while (ccode != L_EXIT);
```

Example 4: *A simple event loop*



A Database System for Automating E-Mail

Storing, receiving, and copying electronic messages

Chris Ohlsen

The proliferation of electronic mail has become an accepted part of today's workplace. That's the good news. The bad news is that in many cases the sheer volume of electronic messages received demands some means of managing them. This article presents a message storage and retrieval system just for this purpose. For the sake of example, the system focuses on MCI Mail, a widely used electronic mail service, and is built around an off-the-shelf database and engine, Paradox from Borland.

For those unfamiliar with MCI Mail, it is an electronic mail service that lets you send messages to individual users (TO:) as well as to a long list of users (CC:). There is an "inbox" for messages sent to you and an "outbox" to save copies of messages you send. MCI also provides a number of advanced features, including those which let you customize your electronic "space" and gain access to other services.

For its part, the Paradox Engine is a database engine with a library of 70 functions callable from Turbo C or Microsoft C. Paradox is a relational database that includes a scripting language called "PAL" (Paradox Application Language) that lets you write within Paradox to manipulate the environment and perform Queries By Example (QBE), fast I/O, and quick prototypes. The inherent flexibility of Paradox and the Paradox Engine makes them ideal for creating an electronic mail storage and retrieval database system.

Before delving too deeply into the system presented here, a quick overview might be useful. As illustrated in Figure 1, your telecommunications program logs on to MCI and prints (and stores on disk) all messages in your MCI "inbox." The telecomm program then logs off MCI. The resulting text file is then parsed by the database application into several Paradox tables. The PAL script presents this information to you, allowing you to respond to a message or create a new one. Next, the database application parses any responses in another table back into a text file. Finally, the telecomm program uploads the responses and new messages to MCI.

Table Structures

To store MCI messages in a Paradox database, you need to develop a data-

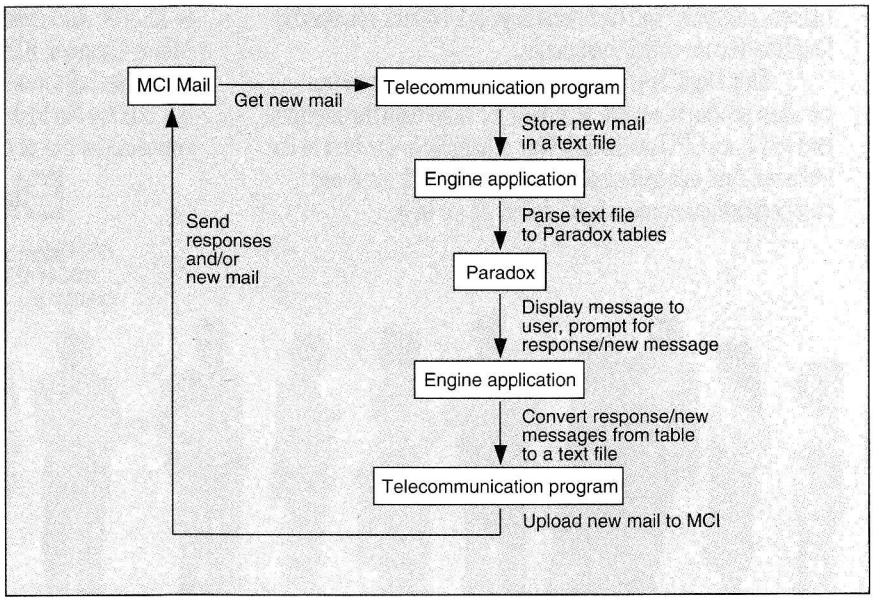


Figure 1: *Flow control diagram of MCI message system*

Chris is an engineer at Borland Intl., specializing in Turbo C++ and the Paradox Engine. He coauthored the book Turbo Pascal Advanced Techniques *(Que, 1989) and can be reached at Borland Intl., P.O. Box 660001, Scotts Valley, CA 95066-0001.*

(continued from page 72)
base structure. This can be done with four tables: the Header, Route, Message, and Pending tables.

The Header table, shown in Table 1(a), includes subject, date received, and name of sender information. With MCI, there are two ways to contact another user. You can use their "user name" (for example, John Smith or JSMITH) or their "MCI address" (for example, 555-1234). The MCI address is *always* unique but the user name may not be; there may be many MCI users named John Smith. (Therefore, if you send mail to a user name, MCI prompts you with a list of all users who have that name and their corresponding unique addresses; you then select the correct John Smith.) However, because the goal here is to develop an automatic system, you should use the MCI address.

Field Name	Field Type
(a) DateReceived	D*
Message#	N*
Subject	A40
From User	A20
From ID	A8
(b) Message#	N*
TOorCC	A5*
User Name	A20*
User ID	A8
(c) Message#	N*
Line#	N*
Text	A80
(d) Pending#	N*
Action	A8*
Text	A40*

Table 1: *Table types: (a) defines the Header table, (b) Routing table, (c) Message table, and (d) Pending table*

Next is the Route table, shown in Table 1(b), which contains the routing information for the message, with a complete list of all the TO: and CC: recipients. The program needs to include all the names if you respond to the message. The Route table lists whether the addressee was included as a TO: or CC: and includes both the user's name and ID.

The Message table, Table 1(c), is the message itself and uses a separate record for each line in the message. The Message# field links the previous two tables with this table. The Line# field allows you to keep the fields in the order that messages are received. The table is displayed in a sorted order based on the Message# field and the Line# field.

Finally, the Pending table, shown in Table 1(d), contains responses to be processed and sent to MCI. The first field is the Pending# field, which links together all of the components of the response. The Action field contains information such as TO:, CC:, Subject, and Filename. The first three strings are associated with the routing and subject matter of the message. The Filename string, however, needs a bit more discussion. Since neither PAL nor C provides the means to edit a message easily, the PAL script runs an external editor. The Filename record contains the name of the ASCII file to be transferred as the body of the message.

The Paradox Engine Application

The storage/retrieval application processes incoming text files and puts the information into the Paradox tables for later use by the PAL script. After the PAL script has run, this application takes the pending message information and translates it into a file that can be uploaded to MCI. However, you must first initialize the database engine, open the tables, and allocate the record buffers. Refer to the C program in Listing One, page 104. (Listing Two, page 104, is the accompanying header file.) *start-*

(continued on page 77)

(continued from page 74)
Engine( ) performs the initialization. A call to *openTables( )* ensures the existence of a table by using the Engine function *PXTblExist( )*. If such a table does not exist, then calls to *PXTblCreate( )* and *PXKeyAdd( )* create the table and indexes. Once these calls are completed, the tables are opened and record buffers allocated.

The storage/retrieval application processes incoming text files and puts the information into the Paradox tables for later use by the PAL script

At this point, note that the *PXCheck( )* function defined in this module takes two parameters: a location and an error code. The location is your current place in the program, and the error code is the return value from the function. This method is a quick and dirty way to trap errors. In cases where an engine function returns a value that we do not want to stop the program for, add the location to a *switch* statement that returns to the caller. The calling code can then check *PXLastErr( )* to determine if there was an unacceptable error in the last call. An example of wanting to know the function's return value is *PXSrchKey( )*. If the key you are searching for does not exist, you do not want to abort the program but continue. The location values are determined as follows: The first digit is the module containing the code; the second is the function containing the code; and the third is the specific call itself.

```
Date:Sun Apr 22, 1990 12:52 pm EST
From:Joe Q. Public / MCI ID: 111-1111

TO:   * Chris Ohlsen / MCI ID: 222-
      2222
TO:   John Smith / MCI ID: 333-3333
TO:   Jane Doe / MCI ID: 444-4444
CC:   Frank Friend / MCI ID: 555-5555
Subject:  Testing MCI program
Hey everybody! This is just a sample
message to see what the layout of
an MCI message looks like.
Joe
```

Figure 2: *Layout of a typical MCI Message*

Parsing Incoming Messages

The program checks the command line for a filename to determine whether there are any incoming files. If there is nothing on the command line, it does not need to process incoming messages. The program will process an outgoing message only if the Pending table exists on the disk. The PAL script creates this table and places all of the outgoing message information in it.

To understand how the program parses incoming messages, study the layout of a typical MCI message in Figure 2 and the parser listed in Listing Three, page 104. The first line of an MCI message contains the Date: string. You can ignore anything that appears before this string in the text file. The program reads the Date: string and parses it using *strtok( )* to remove the components of the date. It then converts these components into data the engine can use, using *PXDateEncode( )*, and adds it to the record for the Header table. The next line of text is the sender information. The program parses both the user's name and MCI ID and adds them to the record.

The next block of information goes to the Routing table. The program uses *getTo( )* and *getCC( )* to add the TO: and CC: lines to the Route table. Every TO: and CC: line is read and parsed to

remove the user's name and MCI ID, and the information is appended to the end of the Route table. Finally, the program reads the Subject line, parsing it and adding it to the record in the Header table.

The script files are pseudo scripts that you can translate into whatever communications package you are using

Once you have all the information for the Header and Route tables, you need to work with the message itself. The program reads the message one line at a time and posts it to the Message table with a unique line number and the current message number. If the line starts with a Date: string, there is another message to process. If the line starts with the Command: string, there are no more messages.

Creating Outgoing Messages

The *send( )* function (see Listing Four, page 106) checks for and processes any outgoing messages. The existence of the Pending table determines if there are any messages to process. The PAL script will create this table only if there is a reply to a message or the user composes a new message. If there are no messages, the function returns. If there is a message, *send( )* continues by opening the table, getting the handles to the fields, and creating an index on the table.

The engine application now enters the main loop of the function, which creates the message script for the communications program. The first line, CR, creates a new message. Then the program addresses the message for both the TO: and CC: fields. Next it adds the Subject: line. The application uses *processMany( )* for each of these fields. It uses a string and the current pending message number as parameters. The program adds these two fields to a temporary record and searches for them in the table. If it finds a match, it reads the record into the Engine's internal buffers, which pull out the desired field and append it to the text file. Then it searches for another match. This continues until there are no matches, at which point the function returns.

(a)

```
Echo Off
comm -sGETMCI.SCR
mci inbox.txt
paradox3 mci
mci
comm -sSENDMCI.SCR
del MCI-SEND.TXT
```

(b)

```
DIAL #-###-###-####     ; Dial your MCI Mail phone number.
WAIT "user name:"
SEND "my mci name"      ; Type in your MCI Account name.
WAIT "Password:"
SEND "my mci password" ; Type in your MCI Password.
WAIT "Command:"
CAPTURE "inbox.txt"     ; Start capturing everything to the file INBOX.TXT.
SEND "pr inbox"         ; Tell MCI to display all message in your Inbox.
WAIT "Command:"
CAPTURE OFF
SEND "exit"             ; Log off of MCI Mail.
EXIT
```

(c)

```
DIAL #-###-###-####          ; Dial your MCI Mail phone number.
WAIT "user name:"
SEND "my mci name"           ; Type in your MCI Account name.
WAIT "Password:"
SEND "my mci password"       ; Type in your MCI Password.
WAIT "Command:"
ASCII XFER NOLF              ; Turn off line feeds during ASCII upload.
UPLOAD ASCII "mci-send.txt" ; Upload all new messages.
```

Example 1: *Sample scripts: (a) MCI.BAT, (b) GETMCI.SCR, and (c) SENDMCI.SCR*

The message itself is stored in an external file. *processFile( )* gets the name of the file from the Pending table, opens the file, and reads it line by line. As it reads each line into the program, *processFile( )* writes out to the message

The Paradox Engine is a database engine with a library of 70 functions callable from Turbo or Microsoft C

script being built. When the end of the file is reached, the message file is closed and erased. *send( )* again gains control and appends a "/" to the end of the message, which tells MCI Mail that the text of the message is ended. *send( )* then adds a blank line to skip through MCI's Handling: prompt, and the "Yes" string is in response to MCI's Send: prompt.

Finally, *nextPendNum( )* gets the number of the next pending function. If there are no more messages pending, it sets the *pendNum* variable to 0. When the loop sees there are no more messages to process, it adds the string "exit\n" to the end of the file, which logs the script off of MCI Mail. The Pending table (and all associated files) is removed from the disk.

At this point, *main( )* takes over again and closes all the open tables, and a call to *PXExit( )* turns the engine off.

The PAL Script

Because PAL cannot compete against your favorite word processor, the MCI PAL script can call an external editor to process the composed messages. The first line of the PAL script (see Listing Five, page 107) defines which editor (*ED*) is called. Make sure the word processor you select is working with an ASCII file without special control characters.

The script then confirms that the tables the engine application has created actually exist. If one of the tables does not exist, the script displays a five-second message on the screen, stating that you must run the MCI.EXE program before running this script.

Next, the script checks the forms to display the Header and Message table information. If the forms do not exist, it will build them. (The Paradox script recording capability allows you to create these functions: Just lay out the fields in the order you want and write down the layout. Then, select Scripts/Begin Record to record the steps.)

The Message table has MultiRecord views so you can display and scroll through the entire message. This form is linked to the Header form. The combined information comprises the view of the individual fields. Notice that the Line# field is decreased to one character width and blocked out by selecting a color because the field is necessary for internal use only. You can change the color selection to match your form display; this block of code is commented in *BuildMsgForms( )*. The third field lets you view the headers of the messages currently stored and select the messages you want to see in full.

Next, an opening screen displays the name of the script and instructions for working with the menus, and the main menu is displayed: Inbox lets you view a message, Compose lets you create a new one, and Quit exits the script. You are prompted for verification if you select Quit.

Viewing the Inbox Messages

The *Inbox( )* function uses the form that was created to display the Header table and lets you choose the message

with which you wish to work. If there are no messages to view, "No messages to be viewed" is displayed on the main screen. If you select Escape, you return to the main menu. If you choose Enter, *ViewMessage( )* selects the message the cursor is currently on.

ViewMessage( ) also uses one of the forms created earlier — the MultiTable form, which was developed with both the Header and Message table. The Message table form has a MultiRecord region that displays the message body. You can use the arrow keys, PgUp, and PgDn to scan through the message. If you want to view the previous or next message in the Header table, press F3 or F4, respectively. These keys correspond with the Up Image and Down Image keys in Paradox. You can also press Escape at this point to return to the main menu.

If you want to respond to the message, press Insert and the script calls *Respond( )*, which grabs the current header information about the subject, message number, and sender. QBE performs a query on the Route table to get all those who received the message, so they can be included in the reply.

Several other fields are set up before pulling the information returned from the query. The Subject is added to the table with the RE: string appended to it to denote a reply. The filename added to the table contains the body of the message. The filename is a combination of the Pending string and an extension of the Pending number in the table.

When these fields are set up, the script scans the answer table built by the previous QBE. It adds all of the fields within the answer table to the Pending table with the appropriate action, indicating a TO: or CC:. DO_IT! saves all of the additions to the Pending table. Finally, the RUN BIG command calls the editor, thus providing the maximum amount of memory.

Composing a New Message

Composing a new message is similar to responding to a prior message except that the information pulled from the Header and Route tables is provided by the user.

At least one TO: field is required to send a message. Therefore, the script forces you to input at least one field. After receiving the TO: fields, it prompts you for the CC: fields. There can be any number of CC:s on a message. Finally, it prompts you for the Subject field. There can be only one of these, too. When all of the information is input, the modified table is saved, and the editor is called with the RUN BIG command.

Pseudo Communication Script and Batch File

Now that you have the major players in the application, the PAL script and the Paradox Engine application, you need something to tie them together. You can use a simple batch file (see Example 1) to call everything, including the pseudo telecommunications package.

The batch file simply needs to call the communications program with a script that will grab all of the available messages on MCI Mail. After it does this, it calls the MCI.EXE program with the name of the capture file to parse and build the tables. Next, it calls Paradox to execute the MCI script. It calls the MCI.EXE application again to parse any responses the script may generate. Finally, it calls the communications script again to upload the message that was created, and the upload file is deleted.

The script files are pseudo scripts that you can translate into whatever communications package you are using. The GETMCI.SCR script in Example 1(b) logs on to MCI Mail and downloads all new messages, capturing the entire session to an Inbox.Txt file. The SENDMCI.SCR script, in Example 1(c) logs on to the service and performs an ASCII upload of the file generated by the MCI.EXE program, MCI-SEND.TXT. The ASCII file creates the message, sends it, and logs off the service.

Conclusions

Many enhancements could be built into the basic system presented here. For one thing, the PAL script could be enhanced to support an address book that keeps track of users' MCI addresses, and the PAL script could add the Route table information to the Address table.

The Paradox Engine application could also be enhanced to do communications internally. It could dial up MCI, post the message, and receive new mail. The program could be set to dial MCI early in the morning before work, and the PAL script could be called so your messages are waiting when you get there. Another possibility is a gateway that would allow many people to use a single MCI Mail account via private "mailboxes." To send a message to one of these people, you could place additional addressing information (such as MBXTO: and MBXCC:) in the body of the message. This extends the application to a network.

DDJ

(Listings begin on page 104.)

Vote for your favorite feature/article.
Circle Reader Service **No. 8.**

Listing One *(Text begins on page 16.)*

```
origin=`hostname`

# run - the user interface component of the shepard system -- B. E. Bauer 1990
# configuration files associated with run:
#   .run.ini    defaults for script,dataset,host,datadir.
#   .current    jobs originating from workstation environment
#   .hosts      host machines able to run shepard
#   .tasklist   list of tasks. Has flags for shepard
#   .runscripts list of machines and possible scripts
#   these files must be located in the login directory
# flag (and task definitions) definitions, Used in case statement
# and passed as actual flag arguments to shepard:
#    x    executes (submits) a job
#    m    monitors job
#    p    probes job
#    s    status of running jobs on all platforms
#    r    list of running jobs
#    k    kill job (with extreme prejudice)
#    t    terminate job in a controlled manner (script dependent)
#    l    list log on remote machine
#    b    bump a waiting job from the .waiting list
#    d    delete a waiting job
#    f    list finished jobs
#    e    list error log
#    c    change host
#    w    list waiting jobs
#    a    list restart jobs
#    g    restart a restartable job

#place the date/time in day-month-year@time single string format
set - `date`
year=$6 month=$2 day=$3 tm=$4
datetime=$day-$month-$year@$tm

echo 'welcome to run on '$origin' at '$datetime

. $HOME/.run.ini    #source the run-script defaults
. $HOME/.shepard.ini # has network definition

# check for finished jobs, update list, display finished list
# find jobs with status RUNNING, check host for status

if (test -f $HOME/.current) then
    cnt=`grep -c DONE $HOME/.current`
    if (test "$cnt" != "0" ) then
        awk 'BEGIN {
            printf "\njobs recently finished\n"
            printf "\n%-10s %-10s %-8s %-21s %-21s\n\n",\
                "script","dataset","host","start","end"
        } $7 == "DONE" {
            printf "%-16s%-16s%-8s%-20s%-20s\n",$1,$2,$3,$5,$6
        } ' $HOME/.current >tmp
        echo ' '; cat tmp   # display list of completed jobs
        e~ho 'press any key to continue \c'; read sel
        cat tmp >> $HOME/run.log   # completed job data to runlog
        awk '$7 != "DONE" {
            print $0
        } ' $HOME/.current >tmp
        mv tmp $HOME/.current
    else
        echo "no new finished jobs"
    fi
fi

# set default host. All activities focus on that host until changed
awk 'BEGIN {
    n=1
    printf "\n----- current hosts -------------------------\n\n"
} {
    if ("'$defhost'" == $1)
        printf "%-3s%-16s%s %s %s  (default)\n",n,$1,$2,$3,$4
    else printf "%-3s%-16s%s %s %s\n",n,$1,$2,$3,$4
    n++
}
END {
    printf "\nselect a host machine by number: "
}' $HOME/.hosts
read sel
if (test -z "$sel") then
    host=$defhost
else
    sel=`awk 'BEGIN {n=1}{if ("'$sel'" == n) print $1; n++}' $HOME/.hosts`
    host=$sel; defhost=$sel
fi
loop=YES

# top of loop. exit with <ret>
while (test "$loop" = "YES")
do
    # display menu of tasks
    echo ' '; echo 'current host is ' $host
    echo ' '
    awk ' BEGIN {
        n=1
        printf "\t#      flag          task\n"
        printf "\t-----------------------------------------------\n"
    }{

        printf "\t%-3s\t%s\n",n,$0
        n++
    }
    END {
        printf "\ntask selection number [<ret> to exit]: "
    } ' $HOME/.tasklist
    read sel
    # look up value for shepard flag associated with task.
    # Use the flag in the case statement
    task=`awk 'BEGIN {n=1} {if("'$sel'" == n) print $2; n++}' $HOME/.tasklist`
    flag=`awk 'BEGIN {n=1} {if("'$sel'" == n) print $1; n++}' $HOME/.tasklist`
```

```
# if response is <ret>, exit while loop
if (test -z "$sel") then
    break
fi

case $flag in
-x) # start a job. Queries for script, dataset, datadir
    # list scripts available only on selected host
    awk ' BEGIN {
        n=1
        def=0
        printf "\n# (host) script"
        printf "\n-------------------------------------\n"
    }
    "'$host'" == $1 {
        if ($2 == "'$defscript'") {
            printf "%-2s %s  (default)\n",n,$0
            def = n
        }
        else printf "%-2s %s\n",n,$0
        n++
    }
    END {
        printf "\nselect a script by number [%s]: ",def
    } ' $HOME/.runscripts
    read tmp
    # look up the script selected by number (must be on one line)
    sel=`awk 'BEGIN{n=1} "'$host'"==$1 {if("'$tmp'" == n) print $2; n++} '
                                                    $HOME/.runscripts`
    if (test "$sel" = "") then
        script=$defscript
    else
        script=$sel; defscript=$sel
    fi
    echo 'selected script is '$script
    # get the dataset name
    echo ' '; echo 'enter dataset name ['$defdata']: \c'
    read sel
    if (test "$sel" = "") then    # substitute default for <ret>
        dataset=$defdat
    else
        dataset=$sel; defdata=$sel
    fi
    echo 'selected dataset is '$dataset
    # get the directory where the data is located
    # if $SHEPARD_NETWORK is set to "remote", data moves between machines
    # using nfs otherwise, data is retained on server
    # home directory on the host machine, then back when done
    echo ' '; echo 'enter directory of data on '
    case $SHEPARD_NETWORK in
        remote) echo $iam': \c';;
           nfs) echo $iam' using nfs mount on '$host': \c';;
        server) echo $host': \c'; defdir='$HOME';;
    esac
    read sel
    if (test "$sel" = "") then    # substitute default for <ret>
        datadir=$defdir
    else
        datadir=$sel; defdir=$sel
    fi
    echo 'selected directory is '$datadir
    # append new job entry to $HOME/.current
    llist='$script $dataset $host $datadir $datetime'
    echo $llist 'out' 'STARTED' >>$HOME/.current
    if (test "$origin" = "$host") then
        shepard $flag $script $dataset $host $datadir
    else
        rsh $host shepard $flag $script $dataset $origin $datadir
    fi;;
-s) # listing of current file. shows activity on other platforms
    awk ' BEGIN {
        fmt="%-5s %-16s %-16s %-21s %-16s\n"
        dash5="-----"
        dash16="----------------"
        dash21="---------------------"
        n=1
        printf "\n\ncurrent job status\n\n"
        printf fmt,"#","script","dataset","submitted","status"
        printf fmt,dash5,dash16,dash16,dash21,dash16
        printf "\n"
    } {
        printf fmt,n,$1,$2,$5,$7
        n++
    }
    END {
        printf "\npress any key to continue "
    } ' $HOME/.current
    read sel;;
-[ktpdbg])
    # these are all list processing commands using pick an item menuing
    # the menu is generated by shepard on the selected host
    # the item is picked in run and the selection happens in shepard
    case $flag in
        -[ktp]) lflag='-r';;  # list running jobs
         -[db]) lflag='-w';;  # list waiting jobs
            -g) lflag='-a';;  # list restartable jobs
    esac
    if (test "$origin" = "$host") then
        shepard $lflag dummy2 dummy3 dummy4 dummy5
    else
        rsh $host shepard $lflag dummy2 dummy3 dummy4 dummy5
    fi
    echo ' '; echo 'select number of job to \c'
    case $flag in
        -k) echo 'kill \c';;
        -t) echo 'halt gracefully \c';;
        -g) echo 'restart \c';;
        -d) echo 'remove from waiting queue \c';;
        -b) echo 'bump to top of queue \c';;
        -p) echo 'probe running status \c';;
```

(continued on page 84)

Listing One *(Listing continued, text begins on page 16.)*

```
        esac
        read sel    # select one from list
        arg5=$sel
        if (test "$origin" = "$host") then
            shepard $flag dummy2 dummy3 dummy4 $arg5
        else
            rsh $host shepard $flag dummy2 dummy3 dummy4 $arg5
        fi;;
    -c) # change hosts
        awk 'BEGIN {
            n=1
            printf "----- current hosts -------------------------\n\n"
        } {
            if ("'$defhost'" == $1) {
                printf "%-3s%-16s%s %s  (default)\n",n,$1,$2,$3,$4 }
            else printf "%-3s%-16s%s %s %s\n",n,$1,$2,$3,$4
            n++
        }
        END {
            printf "select a new host machine by number: "
        }' $HOME/.hosts
        read sel
        if (test -z "$sel") then
            host=$defhost
        else
            sel=`awk 'BEGIN {n=1}{if ("'$sel'"==n) print $1; n++}' $HOME.hosts`
        host=$sel; defhost=$sel
        fi;;
    -[rewfalm]) # process listing commands
        if (test "$origin" = "$host") then
            shepard $flag dummy2 dummy3 dummy4 dummy5
        else
            rsh $host shepard $flag dummy2 dummy3 dummy4 dummy5
        fi
        read sel;;
    *)  # woops
        echo $flag 'is not a recognized option, try again'
    esac
done    # bottom of while loop

# write current values to run-script default file
# $HOME/.run.ini is sourced on invocation in effect restoring the
# last values used. Handy for checking on a previously
# started job - values properly default to the previous
echo 'defscript='$defscript >$HOME/.run.ini
echo 'defdata='$defdata >>$HOME/.run.ini
echo 'defhost='$defhost >>$HOME/.run.ini
echo 'defdir='$defdir >>$HOME/.run.ini

echo 'end of run'
```

End Listing One

Listing Two

```
trap 'rm -f $HOME/.sheplock; exit' 1 2 3 15

# shepard - task management component od shepard system -- B. E. Bauer 1990
# Shepard is the action component of the system. When invoked, it
# owns all the associated files (see top of shepard_queue for list)
# and updates the current file on the originator, log and err files.
# Shepard can be invoked from local or remote machines; it senses
# local or remote operation and behaves accordingly.
# Shepard handles all tasks except for job queueing (shepard_queue) and
# application-specific job probing (defined in $probe_script as sourced
# in 'script'.script). Shepard is called by terminating jobs for cleanup.
# Shepard can be present in several executing copies called by run (the
# user interface) and by completing jobs waiting for cleanup. To avoid
# collision between shepards, absolute ownership of all associated files
# is essential, and is accomplished by creating a lock file. All other
# versions of shepard have to wait until the first is done.

# wait until lock file established insures complete ownership
# of all files by only one version of shepard at a time
until lockon .sheplock
do sleep 5; done

iam=`hostname`
. $HOME/.shepard.ini    # source the initialization file
# do not display greeting message if called from terminating process
if (test "$1" != "-z") then
    echo 'shepard on '$iam' at '`date`
fi
# if you see the message, you made it.
# Important verification that remote shell command is functioning

# lookup values from files depending on mode
pass=NO
case $1 in    # select the file name associated with flag
   -[ktp])    fname=$HOME/.running; pass=YES;;
    -[bd])    fname=$HOME/.waiting; pass=YES;;
      -g)     fname=$HOME/.restart; pass=YES;;
esac
if (test "$pass" = "YES") then # do the lookup
     scr=`awk 'BEGIN {n=1} {if ("'$5'" == n) print $1; n++}' $fname`
    dset=`awk 'BEGIN {n=1} {if ("'$5'" == n) print $2; n++}' $fname`
    host=`awk 'BEGIN {n=1} {if ("'$5'" == n) print $3; n++}' $fname`
    ddir=`awk 'BEGIN {n=1} {if ("'$5'" == n) print $4; n++}' $fname`
     sel=`awk 'BEGIN {n=1} {if ("'$5'" == n) print $5; n++}' $fname`
   tname=$host':'$scr'('$dset')'        # compact file name
fi

# no loop in shepard. Does the command then exits
case $1 in
    -x) # runs job through queue manager which handles submission
        shepard_queue $2 $3 $4 $5;;
```

(continued on page 86)

Listing Two *(Listing continued, text begins on page 16.)*

```
-m) # system-dependent code here. "big" is using Berkeley UNIX
    # while all others use SYSTEM V. Options to ps are different
    if (test "`hostname`" = "big") then
        ps -ax ¦ grep -n shepard_exec  # CONVEX specific (for example)
    else
        ps -ef      # SGI IRIS specific (for example)
    fi;;
-p) # probe job - script-dependent
    # source the file containing application-specific scripts
    . $SHEPARD_DIR/$2.script
    . $probe_script;;    # defined in sourced file $script.script
    echo 'press <ret> to continue \c'
-r) # list running jobs on host
    cnt=`wc -l $HOME/.running ¦ awk '{print $1}'`
    if (test "$cnt" = "0") then
        echo ' '; echo 'no jobs running'; echo ' '
    else
        awk ' BEGIN {
            fmt="\n%-5s %-16s %-16s %-8s\n"
            printf "\n----- running jobs on %s -----\n","'$host'"
            printf fmt,"#","script","dataset","pid"
            printf "----- ---------------- ---------------- --------\n"
            n=1
        } {
            printf fmt,n,$1,$2,$5
            n++
        }
        END {
            printf "\npress any key to continue "
        } ' $HOME/.running
    fi;;
-w) # list waiting jobs on host
    cnt=`wc -l $HOME/.waiting ¦ awk '{print $1}'`
    if (test "$cnt" = "0") then
        echo ' '; echo 'no jobs waiting'; echo ' '
    else
        awk ' BEGIN {
            fmt="\n%-5s %-16s %-16s %-8s\n"
            printf "\n----- waiting jobs on %s -----\n","'$host'"
            printf fmt,"#","script","dataset","position"
            printf "----- ---------------- ---------------- --------\n"
            n=1
        } {
            printf fmt,n,$1,$2,$5
            n++
        }
        END {
            printf "\npress any key to continue "
        } ' $HOME/.waiting
    fi;;
-a) # list restartable jobs on host
    cnt=`wc -l $HOME/.restart ¦ awk '{print $1}'`
    if (test "$cnt" = "0") then
        echo ' '; echo 'no jobs in restart'; echo ' '
    else
        awk ' BEGIN {
            fmt="\n%-5s %-16s %-16s %-8s\n"
            printf "\n----- restartable jobs on %s -----\n","'$host'"
            printf fmt,"#","script","dataset","position"
            printf "----- ---------------- ---------------- --------\n"
            n=1
        } {
            printf fmt,n,$1,$2,$5
            n++
        } ' $HOME/.restart
    fi;;
-g) # restart a job from $HOME/.restart and update
    # file to select passed as shell argument 5
    # copys the selected entry to $HOME/.waiting with priority=RESTART
    awk ' BEGIN {
        n=1
    } {
        if (n == "'$5'") printf "%s %s %s %s RESTART\n",$1,$2,$3,$4
        n++
    } ' $HOME/.restart >> $HOME/.waiting
    awk ' BEGIN {   # restarted job is purged from $HOME/.restart
         n=1
    } {
        if (n != "'$5'") print $0
        n++
    }' $HOME/.restart > tmp
    mv tmp $HOME/.restart
    echo 'restarting '$tname' at '$datetime >>shepard.log
    #update .current on origin machine
    if (test "$host" = "$iam") then
        run_update -g $scr $dset $sel
    else
        rsh $host run_update -g $scr $dset $sel
    fi
    shepard_queue -r;;   # do the restart
-k) # kill job with extreme prejudice
    # pid passed as shell argument 5, assigned to sel
    # running processes have 2 entries in the process list
    # first = shepard_exec and has the pid stored in running
    # second = the executable application
    # searching the process list for first finds second; both
    # must be killed to stop the application: killing shepard_exec
    # alone leaves the application program still running
    if (test "$iam" = "big") then
        cleanup=`ps -axl ¦ awk ' "'$sel'" == $4 {print $3}'`
    else
        cleanup=`ps -ef ¦ awk ' "'$sel'" == $4 {print $3}'`
    fi
    kill -9 $sel
    kill -9 $cleanup
    if (test "$?" = "0") then
        echo 'killed '$tname' at '$datetime >>$HOME/shepard.log
    else
        echo 'status of kill command nonzero - check log for problems'
    fi
```

```
        awk ' $5 != "'$sel'" { print $0 }' $HOME/.running > $HOME/tmp
        mv $HOME/tmp $HOME/.running
        #update .current on origin machine
        if (test "$host" = "$iam") then
            run_update -k $scr $dset $sel
        else
            rsh $host run_update -k $scr $dset $sel
        fi
        shepard_queue -q;; # check for waiting jobs
    -t) # terminate job gracefully pass script and origin variables
        # source the file containing application-specific scripts
        . $SHEPARD_DIR/$2.script
        . $terminate_script # found in scriptname.script
        echo 'terminated '$tname' at '$datetime >> $HOME/shepard.log
        #update .current on origin machine
        if (test "$host" = "$iam") then
            run_update -t $scr $dset $sel
        else
            rsh $host run_update -t $scr $dset $sel
        fi;;  # when the application exits, it will check for waiting jobs
    -l) # list the job log on host
        tail -30 shepard.log;;   # only the last is generally interesting
    -b) # bump priority of specific job
        # $HOME/.waiting can be in any order, use 2-pass approach
        # pass 1: set desired to zero, increment all others
        # pass 2: change 0 to 1, zero now being easy to spot
        awk ' {
            if ($1 == "'$scr'") {
                if ($5=="'$sel'") $5 = 0
                if ($5 < "'$sel'") $5 += 1
            }
            printf "%s %s %s %s %s\n",$1,$2,$3,$4,$5
        } ' $HOME/.waiting | awk ' {
                if ($5 == 0) $5 = 1
                printf "%s %s %s %s %s\n",$1,$2,$3,$4,$5
        } ' > $HOME/tmp
        mv $HOME/tmp $HOME/.waiting
        echo 'bumped '$tname' at '$datetime >> $HOME/shepard.log
        if (test "$host" = "$iam") then
            run_update -b $scr $dset $sel
        else
            rsh $host run_update -b $scr $dset $sel
        fi;;
    -d) # delete a waiting job from waiting, selected passed as shell arg 5
        # same script/higher priority have their priorities--
        awk ' {
            if ($1 == "'$scr'") {
                if ($5=="'$sel'") next   # excise deleted job
                if ($5 > "'$sel'") $5 = $5 - 1
            }
            print $0
        } ' $HOME/.waiting > tmp
        mv tmp $HOME/.waiting
        echo 'deleted '$tname' at '$datetime >> $HOME/shepard.log
        #update .current on origin machine
        if (test "$host" = "$iam") then
            run_update -d $scr $dset $sel
        else
            rsh $host run_update -d $scr $dset $sel
        fi;;
    -f) # list finished jobs
        awk ' BEGIN {
            fmt="\n%-16s %-16s %-12s\n"
            printf "\n----- finished jobs on %s -----\n","'$host'"
            printf fmt,"script","dataset","origin"
            printf "---------------- ---------------- ------------\n"
        } {
            printf fmt,$1,$2,$3
        }
        END {
            printf "\npress any key to continue "
        } ' $HOME/.finished;;
    -e) # list error log
        tail -30 $HOME/shepard.err;;
    -z) # go to cleanup routine, $5 has the completed jobs pid number
        echo 'finished '$4':'$2'('$3') at '`date` >>shepard.log
        # write entry to .finished
        # run on origin will look here for completed jobs
        echo $2 $3 $4 $5 `date` >> $HOME/.finished
        # excise finished job from $HOME/.running list
        awk '{
            if ("'$5'" != $5) print $0
            }' $HOME/.running >tmp
        mv tmp $HOME/.running
        #update .current on origin machine
        if (test "$4" = "$iam") then
            run_update -f $2 $3 $5
        else
            rsh $4 run_update -f $2 $3 $5
        fi
        #check queue for waiting process
        shepard_queue -q;;
esac

rm -f $HOME/.sheplock     # remove locking file

# normal return to run if invoked by remote shell, otherwise terminates
```

End Listing Two

Listing Three

```
trap 'rm -f $HOME/tmp; exit' 1 2 3 15

# shepard_queue - queue manager for shepard system -- B. E. Bauer 1990
# shepard_queue places jobs in a waiting queue and allows a job
# to actually start if the count of similar jobs running is
# below a user defined threshold. Its like a FIFO queue with a twist.
# This is intended to balance throughput vs system demands on
# multiprocessor high performance computers. Alter for your environment
```

(continued on page 88)

Listing Three *(Listing continued, text begins on page 16.)*

```
# jobs in $HOME/.waiting have a number associated with their place in the
# queue. 1=next to start up to limit defined in .limits
# passed arguments:
#   normal queue submit:    1:  script name
#                           2:  dataset name
#                           3:  originating machine name
#                           4:  dataset directory
#   restart                 1:  -r  (no other values passed)
#   queue check             1:  -q  (no other values passed)
#
#   for restart, $HOME/.waiting has the restart job preappended
iam=`hostname`
. $HOME/.shepard.ini    # source the initialization file
mode=NORMAL
if (test "$1" = "-r") then    # restart entry submitted
    # get the script which has the RESTART code (normally passed as $1)
    scr=`awk 'BEGIN {n=0} $5=="RESTART" {print $1}' $HOME/.waiting`
    # find and replace RESTART with last queue slot for corresponding script
    awk ' BEGIN {
        count = 1
    } {
        if ("'$scr'" != $1) print $0
        else if ($5 != "RESTART") {
            count++
            printf "%s %s %s %s %s\n",$1,$2,$3,$4,$5
        }
        else printf "%s %s %s %s %s\n",$1,$2,$3,$4,count
    } ' $HOME/.waiting  > $HOME/tmp
    mv $HOME/tmp $HOME/.waiting
elif (test "$1" != "-q") then   # new job to submit
    # append new job entry to $HOME/.waiting list
    echo $1 $2 $3 $4 'NEW' >> $HOME/.waiting
    # change NEW label to count of jobs having that script
    # newest entry has the highest number/last to be executed
    awk ' BEGIN {
        count = 1
    } {
        if ("'$1'" != $1) print $0
        else if ($5 != "NEW") {
            count++
            print $0
        }
        else printf "%s %s %s %s %s\n",$1,$2,$3,$4,count
    } ' $HOME/.waiting  > $HOME/tmp
    mv $HOME/tmp $HOME/.waiting
    cnt=`awk 'BEGIN{n=0}"'$1'" == $1 {n++} END {print n}' $HOME/.waiting`
    if (test "$3" = "$iam") then
        run_update -w $1 $2 cnt
    else
        rsh $3 run_update -w $1 $2 cnt
    fi
else
    mode=QUEUE    # flag suppresses terminal response when in -q mode
fi

didit=NO  # flag reports job starting status

# loop through scripts available on this host
# available scripts are in the environment variable SHEPARD_SCRIPTS
# The FIFO queue has a twist: differing job types are subqueued with
# limits for each found in .limits without maintaining separate queue
# structures. This method is easier to implement and permits a maximum
# load balance consisting of a mix of program types, tailored to ones
# needs. In this way, a number of program type 'a' exceeding the limit
# only runs the number set in .limits, while the others queue leaving
# processor time for program types 'b' and 'c'. The optimum load balance is
# determined by the system resource requirements of each program and
# ones needs for throughput; adjusting .limits allows changes on the fly.

for i in $SHEPARD_SCRIPTS
do
    # count jobs actually running for each script, get associated job limit
    if (test -f "$HOME/.running") then
        rcnt=`awk 'BEGIN{n=0} "'$i'"==$1 {n++} END{print n}' $HOME/.running`
    else
        rcnt=0  # set rcnt to 0 if $HOME/.running is not present
    fi
    rlim=`awk '"'$iam'" == $1 && "'$i'" == $2 {print $3}' $HOME/.limits`
    if (test -z "$rlim") then
        rlim=1  # if no limit in $HOME/.limits, one job permitted
    fi
    # if more running jobs exceeds the limit, continue to next script
    if (test "$rcnt" -ge "$rlim") then
        continue
    fi
    # loop to next script if no jobs waiting with priority=1
    script=`awk ' "'$i'" == $1 && $5 == "1" { print $1}' $HOME/.waiting`
    if (test "$script" != "$i") then
        continue
    fi
    # found one for current script, get the remaining values
    dataset=`awk ' "'$i'" == $1 && $5 == "1" { print $2}' $HOME/.waiting`
    origin=`awk ' "'$i'" == $1 && $5 == "1" { print $3}' $HOME/.waiting`
    datadir=`awk ' "'$i'" == $1 && $5 == "1" { print $4}' $HOME/.waiting`

    # put date/time in a single string format
    set - `date`
    day=$3 month=$2 year=$6 tm=$4
    datetime=$day-$month-$year@$tm
    # submit shepard_exec to the background, get its pid
    wait 10    # shepard_queue does not wait for shepard_exec
    nohup shepard_exec $script $dataset $origin $datadir >shepard_junk.log &
    pid=$!  # process identification number - unique for job
    errflag=$?

    # shepard_exec did not initiate, for some reason
    # append the shepard_junk.log to shepard.err, alert the user
    # the job is placed in .restart
    if (test "$errflag" != "0") then
        #notify the user
        echo $script'('$dataset') did not start at '$datetime
        echo '       return code '$errflag
        echo '------ process error logfile contents -----'
        cat $HOME/shepard_junk.log
        echo '------ end of log from '$script'('$dataset') -----'
        echo ' '; echo 'check the contents of shepard.err  for details'
        # update shepard.err
        echo $script'('$dataset') did not start at '$datetime >tmp
        echo '------ process error logfile contents -----' >>tmp
        cat shepard_junk.log >>tmp
        echo '------ end of log from '$script'('$dataset') -----' >>tmp
        cat tmp >> $HOME/shepard.log; rm tmp
        # remove from $HOME/.waiting, place in .restart
        awk ' $1 == "'$script'" && $2 == "'$dataset'" && $5 == 1 {
            print $0
        }' $HOME/.waiting >> $HOME/.restart
        awk '{
            if ($1 == "'$i'" && $5 == 1) continue
            if ($1 == "'$i'") {
                $5 = $5 - 1
                printf "%s %s %s %s %s\n",$1,$2,$3,$4,$5
            }
        }'    $HOME/.waiting > $HOME/tmp
        mv $HOME/tmp $HOME/.waiting
        exit
    fi

    didit=YES
    # append job specifics to $HOME/.running
    echo $script $dataset $origin $datadir $pid >>$HOME/.running

    # append job info to shepard.log
    echo $script'('$dataset') started '$datetime >>$HOME/shepard.log

    # remove running job from $HOME/.waiting, update priority
    awk '{
        if ($1 == "'$i'" && $5 == 1) next
        if ($1 == "'$i'") {
            $5 = $5 - 1
            printf "%s %s %s %s %s\n",$1,$2,$3,$4,$5
        }
    }'    $HOME/.waiting > $HOME/tmp
    mv $HOME/tmp $HOME/.waiting
    #update .current on origin machine
    if (test "$iam" = "$origin") then
        run_update -r $script $dataset $pid
    else
        rsh $origin run_update -r $script $dataset $pid
    fi
    # if job is successfully started, notify user
    if (test "$didit" = "YES") then
        echo ' '
        echo $script'('$dataset') started on '$iam' at '$datetime
    fi
done

if (test "$didit" = "NO") then
    if (test "$mode" != "QUEUE") then
        echo ' '; echo 'no jobs were submitted'
    fi
fi
trap '' 1 2 3 15
```

End Listing Three

Listing Four

```
trap 'shepard -z $1 $2 $3 $$' 1 2 3 15

# shepard_exec - execution potion of shepard system -- B. E. Bauer, 1990
# passed args: 1: script, 2: dataset, 3: origin, 4: datadir

. $HOME/.shepard.ini      # source initialization file
. $SHEPARD_DIR/$1.script  # source application-specific definitions

# routine to move the required files into the execution environment
# sourcing vs separate shell obviates need to pass values
. $SHEPARD_DIR/$getdata_script

# run the program. Assumes here that stdin, stdout, and stderr are
# required (generally true for UNIX) during execution. All other data
# files were moved into the execution environment by $getdata_script
$exe < $2$inp 1> $2$log 2> $2$err

# source the script to return data back in its proper location
. $SHEPARD_DIR/$putdata_script

# clean up and update status files
shepard -z $1 $2 $3 $$  # pid of completing process returns as arg5

trap '' 1 2 3 15
```

End Listing Four

Listing Five

```
trap 'rm -f $HOME/tmp; exit' 1 2 3 15

#   run_update - update component of shepard system -- B.E. Bauer 1990
#   updates the .current file to reflect system activities

flag=$1 script=$2 dataset=$3 opt=$4
set - `date`
day=$3 month=$2 year=$6 tm=$4
datetime=$day-$month-$year@$tm

case $flag in
    -w) stat='WAITING';;
    -r) stat='RUNNING';;
    -g) stat='RESTART';;
    -b) stat='BUMPED';;
    -k) stat='KILLED';;
    -f) stat='DONE';;
```

(continued on page 90)

Listing Five *(Listing continued, text begins on page 16.)*

```
    -d) stat='DELETED';;
    -t) stat='TERMINATED';;
esac

awk ' {
    if ("'$script'" == $1 && "'$dataset'" == $2) {
        printf "%s %s %s %s %s %s %s\n",$1,$2,$3,$4,$5,"'$datetime'","'$stat'"
    }
    else print $0
}' $HOME/.current >$HOME/tmp
mv $HOME/tmp $HOME/.current

trap '' 1 2 3 15
```

End Listing Five

Listing Six

```
big bmin311v     Batchmin large  (<2000 atoms)
big bmin31mv     Batchmin medium (<1000 atoms)
big spartan      ab initio electronic structure calculation
big amber        biological structure simulation
big smapps       Monte Carlo peptide simulation
big ampac        semi-empirical electronic structure calculation
big dspace       NMR distance -> structure
moe bmin311s     Batchmin large  (<2000 atoms) use with caution!
moe bmin31ms     Batchmin medium (<1000 atoms) default
moe bmin31ss     Batchmin small  (<250 atoms)
moe amber        biological structure simulation
moe ampac        semi-empirical electronic structure calculation
moe spartan      ab initio electronic structure calculation
moe smapps       Monte Carlo peptide simulation
larry bmin31ss   Batchmin small (<250 atoms)
larry ampac      semi-empirical electronic structure calculation
```

End Listing Six

Listing Seven

```
larry SGI IRIS 4D/25TG (B-1-3-09)
curly SGI IRIS 4D/120GTX (B-8-3-22; CADD room)
moe SGI IRIS 4D/240s (B-8-3-22; CADD room)
big CONVEX 220 (B-20-B)
```

End Listing Seven

Listing Eight

```
-x execute (submit) a job
-m monitor remote host process status (ps command)
-p probe running job
-s status of running jobs on all platforms
-r running job list
-k kill job (with extreme prejudice)
-t terminate job gracefully
-g restart job
-l log file on remote machine (tail -30)
-b bump waiting job to next
-d delete a waiting job
-f finished job list on host machine
-e error log on host machine (tail -30)
-c change hosts
-w waiting job list on host machine
-a restartable job list on host machine
```

End Listing Eight

Listing Nine

```
# Loads values for script, dataset, host, and datadir last used by run.
# This file is recreated at the end of run.

defscript=bmin311v
defdata=bmintest3
defhost=big
defdir=pla2
```

End Listing Nine

Listing Ten

```
big bmin311v 3
big bmin31mv 1
big spartan 1
big amber 2
big smapps 1
big ampac 3
moe bmin31ms 2
moe bmin31ss 4
moe amber 2
moe smapps 1
moe ampac 4
moe spartan 1
larry bmin31ss 1
larry ampac 1
```

End Listing Ten

Listing Eleven

```
# Definitions for runnable scripts, dataset network movement and directory for
# various files. The runnable scripts must be in agreement with contents of
# .runscripts. Behavior of network for the originating machine is set here.

# options for SHEPARD_NETWORK: server, nfs, remote
# SHEPARD_DIR is location of application-specific shepard scripts

# this file is sourced and executes directly in environment of script

case `hostname` in
    larry|curly) # SGI IRIS workstation definitions
        SHEPARD_SCRIPTS='bmin31ss ampac'
        SHEPARD_NETWORK=server
        SHEPARD_DIR=$HOME/shepard_dir;;
    moe) # SGI IRIS-240 compute server definitions
        SHEPARD_SCRIPTS='bmin31ss bmin31ms amber smapps ampac spartan'
        SHEPARD_NETWORK=server
        SHEPARD_DIR=$HOME/shepard_dir;;
    big) # CONVEX specific definitions
        SHEPARD_SCRIPTS='bmin311v bmin31mv amber smapps ampac spartan dspace'
        SHEPARD_NETWORK=server
        SHEPARD_DIR=$HOME/shepard_dir;;
esac
```

End Listing Eleven

Listing Twelve

```
# bmin311v script for large vector (CONVEX) version of Batchmin v 3.1
exe=bmin311v                          # the executable (in PATH)
inp=.com                              # extension for standard input
log=.log                              # extension for standard output
err=.err                              # extension for error output (channel 2)
getdata_script=bmin311v.getdata       # get the input datafiles
putdata_script=bmin311v.putdata       # put the output back
terminate_script=bmin311v.terminate   # application-specific shutdown
probe_script=bmin311v.probe           # conducts an application-specific probe
```

End Listing Twelve

Listing Thirteen

```
# bmin311v.getdata: get data script. Sourced in shepard_exec
# shell args: 2: dataset, 3:origin, 4: datadir
# datadir is dependent on network choice:
#       server: host-$HOME/datadir (host-$HOME is prepended)
#          nfs: nfs path of data from host to origin machines
#       remote: origin-$HOME/datadir (origin-$HOME is prepended)

case $SHEPARD_NETWORK in
    server) cd $4;;     # data stays put on host machine
    remote) rsh $3 cat $4/$2.dat >$2.dat   # move data. This is a kluge
            rsh $3 cat $4/$2$inp >$2$inp;; # remote cat puts output on host
       nfs) cp $4/$2.dat .     # copy via remotely mounted nfs dir
            cp $4/$2$inp . ;;
esac
```

End Listing Thirteen

Listing Fourteen

```
# bmin311v.putdata: put data script. Sourced in shepard_exec
# shell args: 2: dataset, 3:origin, 4: datadir
# datadir is dependent on network choice:
#       server: host-$HOME/datadir (host-$HOME is prepended)
#          nfs: nfs path of data from host to origin machines
#       remote: origin-$HOME/datadir (origin-$HOME is prepended)

# all application-specific output files are moved, if necessary
case $SHEPARD_NETWORK in
    server) cd $HOME;;   # movement of files is not necessary
    remote) rsh $3 cat ">"$4/$2.out <$2.out     # another network kluge
            rsh $3 cat ">"$4/$2$log <$2$log     # remote cat with ">" writes
            rsh $3 cat ">"$4/$2$err <$2$err;;   # to remote. < read local.
       nfs) cp $2.out $4/$2.out
            cp $2$log $4/$2$log
            cp $2$err $4/$2$err;;
esac
```

End Listing Fourteen

Listing Fifteen

```
# sourced from shepard
# batchmin terminates when it finds dataset.stp in execution dir

case $SHEPARD_NETWORK in
        server) echo 'help me, please help me' > $4/$2.stp;;
    remote|nfs) echo 'help me, please help me' > $2.stp;;
esac
```

End Listing Fifteen

Listing Sixteen

```
# Sourced from shepard. $dset=jobname, $ddir=directory, $log=logfile ext
# Script prints the last 30 lines of the log file from the selected job

case $SHEPARD_NETWORK in
        server) tail -30 $ddir/$dset$log;;
    remote|nfs) tail -30 $dset$log;;
esac
```

End Listing Sixteen

Listing Seventeen

```
/* lockon.c - creates lock file from argv[1] having no privelege
   B. E. Bauer, 1990
 */
main (argc, argv)
int argc;
char *argv[];
{
    int fp, locked;

    locked = 0;
    if (argc != 1) {
        printf ("\nuseage: lockon lockfile\n");
        exit (0);
    }
    if ((fp = creat(argv[1], 0)) < 0) ++locked;
    else close(fp);
    return (locked);
}
```

End Listings

Listing One *(Text begins on page 24.)*

```
/*########## INITIALIZE INTERRUPTS, X.25 & TRANSPORT LAYER DATA #########*/
/* AUTHOR: Michael T. Thompson, Planning Systems Inc. */
/*          2-16-89 for Mitre Corp. (W85)  */

/* initialize interrupt routines */

#include <stdio.h>      /* MicroSoft "C" 5.1 " */
#include <dos.h>        /* MS-DOS 3.30 */
#include <ctype.h>

extern interrupt far clock();   /* Retix clock routine */
extern interrupt far rcvdata(); /* Interrupt interface routine */
extern buf_type bfh_head;
extern unsigned char *rcvdat;  /* Common Receiver buffer pointer varible */
extern buf_type rcvbuf;        /* Receiver User Data buffer pointer */
extern int icnt, rbufcnt;      /* Number of pointers not used in Baddr Array */
extern buf_type Baddr[];       /* Array of buffer pointers */

int xbufsiz=256;             /* Standard Buffer size */
int number_of_buffers=30;    /* Total number of transmit and receive buffers */

/* ----- Primary system Initialization routine ----- */
x25_init()
{
int i, tbufsiz, result;         /* Local varibles */
tbufsiz = xbufsiz + 30;         /* Buffer size plus header data */

        _disable();         /* disable interrupts while initializing */
        init_memory();      /* get available memory from DOS */
        init_bufpool(&bpool,tbufsiz,number_of_buffers); /* setup buffer pool */
        init_timers();      /* initialize system timers */
        init_rcvbuf();      /* initialize receive buffers */
        cominz();           /* initialize clock and I/O Interrupt routines */
        _enable();          /* start interrupts backup */
}

/*------------------- Initialize Receive Buffer Array --------------------*/
init_rcvbuf()
{
int i;

        for(i=0;i<10;i++)               /* store pointers in buffer array */
           Baddr[i] = getbuf(&bpool,xbufsiz+30); /* +30 for X.25 header info */
        rbufcnt = 9;                    /* 0 to 9 = 10 buffers */
        rcvbuf = Baddr[rbufcnt];        /* preallocate first buffer pointer */
        Baddr[rbufcnt] = (buf_type) NULL;     /* clear the array pointer */
        rcvdat = (char *)(BuffData(rcvbuf));  /* point to the user space */
        icnt = 0;                       /* zero frame character count */
}

/* ######################## COMINZ.C ########################### */
/*   Initialize the Clock and Synchronous Interrupt routines    */
cominz()
{
unsigned intnum;
unsigned int val;       /* local varibles */

        /* install MAC I/O driver */
        intnum = 0x1c;                  /* Clock interrupt vector */
        _dos_setvect(intnum,clock);     /* setup new clock interrupt routine */
        istart();                       /* enable I/O interrupt processing */
        intnum = 0x0c;                  /* I/O interrupt vector */
        _dos_setvect(intnum,rcvdata);   /* setup tx & rx interrupt routine */
        val = inp(0x21);
        outp(0x21,(val & 0xc7));        /* start irq 3, 4, & 5 */
}

/* ++++++++++++++++ INITIALIZE SEALEVEL BOARD ++++++++++++++++++ */
istart()
{
unsigned char InitArray[50];    /* Allocate the size of the Init Array */
int iCount;
        /*------------------- Section # 1 ----------------------*/
        InitArray[0]  = 9;
        InitArray[1]  = 0xC0;   /* force hardware reset */
        InitArray[2]  = 0;
        InitArray[3]  = 0x00;
        InitArray[4]  = 4;
        InitArray[5]  = 0x20;   /* (SDLC) mode selected */
        InitArray[6]  = 3;
        InitArray[7]  = 0xC0;   /* rx 8 bits, sync char inhabit */
        InitArray[8]  = 5;
        InitArray[9]  = 0x61;   /* tx 8 bits */
        InitArray[10] = 6;
        InitArray[11] = 0;      /* For Mono-sync */
        InitArray[12] = 7;
        InitArray[13] = 0x7E;   /* sdlc sync character */
        InitArray[14] = 10;
        InitArray[15] = 0x80;   /* CRC set to inverted bit pattern */
        InitArray[16] = 11;
        InitArray[17] = 0;      /* for dte clock from RTxC pin */
        InitArray[18] = 12;
        InitArray[19] = 0xFE;   /* Low order baud rate value */
        InitArray[20] = 13;
        InitArray[21] = 00;     /* High order baud rate value */
                                /*                          H  L  */
                                /*              38400 = 00 3E  */
                                /*              19200 = 00 7E  */
                                /* time constant: 9600 = 00 FE  */
                                /*               4800 = 01 FE  */
        InitArray[22] = 14;
        InitArray[23] = 2;      /* gen enabled, gen source */
        /*------------------- Section # 2 ----------------------*/
        InitArray[24] = 14;
        InitArray[25] = 3;      /* gen enabled, gen source */
        InitArray[26] = 3;
        InitArray[27] = 0xD9;   /* rx 8 bits, hunt mode, rx CRC, rx enabled */
        InitArray[28] = 5;
        InitArray[29] = 0x69;   /* tx 8 bits, tx enabled, tx CRC */
        InitArray[30] = 0;
        InitArray[31] = 0x80;   /* tx CRC gen */
        InitArray[32] = 1;      /* int on all rx chars or special cond. */
        InitArray[33] = 0x12;   /* enable tx interrupts */
        /*------------------- Section # 3 ----------------------*/
        InitArray[34] = 15;
        InitArray[35] = 0x41;   /* tx underrun int enabled */
        InitArray[36] = 0;
        InitArray[37] = 0x30;   /* error reset */
        InitArray[38] = 0;
        InitArray[39] = 0x90;   /* tx CRC gen, reset ext/status int */
        InitArray[40] = 0;
        InitArray[41] = 0x90;   /* twice */
        InitArray[42] = 1;      /* int on all rx chars or special cond. */
        InitArray[43] = 0x12;   /* enable tx interrupts */
        InitArray[44] = 9;
        InitArray[45] = 8;      /* Master int enable */
        InitArray[46] = 0;
        InitArray[47] = 0xF0;   /* reset tx underrun, error reset */
        InitArray[48] = 0;
        InitArray[49] = 0x28;   /* reset tx int pending */

        for (iCount = 0; iCount < 50; iCount++)
                outp(0x239,InitArray[iCount]);  /* Output Data to SDLC Chip */
```

End Listing One

Listing Two

```
/*****************************************************************
 *  RCV2.C -- Interrupt Handler routine for use with the Sealevel   *
 *  Systems synchronous communications board that uses preallocated *
 *  buffers to Transmitt and Receive X.25 data frames.              *
 *****************************************************************/

/* AUTHORS - Michael T. Thompson, Planning Systems Inc. and
 *           Ken Crocker, The MITRE Corporation 5/23/89
 */

#include <stdio.h>      /* MicroSoft "C" 5.1 " */
#include <dos.h>                /* MS-DOS 3.30 */
#include <ctype.h>

/* RETIX OSI SOFTWARE COMMON HEADER FILES */
#include "c:\retix\include\bufflib.h"
#include "c:\retix\include\common.h"
#include "c:\retix\include\system.h"
#include "c:\retix\include\lapb.h"
#include "c:\retix\include\address.h"
#include "c:\retix\include\network.h"
#include "c:\retix\include\x25.h"

extern buf_type MDATcon();      /* Data Confirm routine */
extern struct sp_ent *mac;      /* Service provider table */
extern xbufsiz;                 /* Current TPDU buffer size */
extern vpmidi();                /* MDATind Data Indiction routine */

unsigned char *rcvdat;        /* Pointer to received user data input buffer */
buf_type rcvbuf;              /* Pointer to received buffer header */
buf_type Baddr[10];           /* Array of allocated receive buffer pointers */
int icnt, rbufcnt, fcnt;      /* Receiver and Transmitter counters */
int fsize;                    /* Size of Transmit Frame */
char *frame;                  /* Temporary pointer to Transmit frame */

/************************* RCVDATA.C ***************************
 *          Interrupt Driven Receiver and Transmitter          *
 *  RECEIVER: On the receive side an array of buffer pointers is *
 *  allocated in the x251t.c initialization Routine. The first   *
 *  buffer is preassigned to the receive routine and then that   *
 *  buffer can be written to by the interrupt routine. When the  *
 *  last character of the frame is received the MDATind (vpmidi) *
 *  routine is called and the new buffer is put on the queue for *
 *  processing later. After that we get the address of another   *
 *  preallocated buffer from the array and setup the proper      *
 *  pointers.                                                    *
 *****************************************************************/

void interrupt cdecl rcvdata()
{
unsigned int c, c1, delay;      /* local varibles */
_enable();                      /* enable interrupts */

/* check if this is an error, a receive or a transmitt interrupt */
outp(0x239,3);
delay = 0;                  /* allow time for the register to setup */
   if(((c = inp(0x239)) & 0x30) != 0)  /* if not an error continue */
        {
                if((c & 0x20) != 0)     /* if not receive interrupt continue */
                {

        /******** RECEIVE DRIVER *********/
        /*   we must have receive data   */
        /***********************************/
        rcvdat[icnt++] = inp(0x238);    /* get character and store it */
        outp(0x239,1);
        delay = 0;
        if(((c1 = inp(0x239)) & 0x80) != 0)     /* check for end of frame */
        {
                if((c1 & 0x70) == 0)  /* check for a valid CRC */
                {       /* must be OK */
```

```
                if(icnt > 3)
                {
                  rcvdat[icnt-2] = 0;
                  BuffAdjust(rcvbuf,((xbufsiz+30)-(icnt-2)));
                     /* reajust buffer size for lapb */
                  vpmidi(rcvbuf,mac);
                    /* send MDATind to service user */
                  icnt = 0;
                  while(Baddr[--rbufcnt] == (buf_type) NULL);
                    /* Find an unused buffer pointer */
                  rcvbuf = Baddr[rbufcnt];
                    /* got a new receive buffer pointer */
                  Baddr[rbufcnt] = (buf_type) NULL;
                    /* clear buffer pointer from array */
                  rcvdat = (char *)(BuffData(rcvbuf));
                    /* setup receive data pointer */
                    /* check for a transmit interrupt pending */
                  if((c & 0x10) == 0)
                    goto ENDINT;  /* no interrupt so end */
                /* found transmit interrupt pending so send it */
                     goto TXINT;
                }
                /* frame is to short so go to error reset */
                goto ERRRES;
        }
        /* We got a BAD CRC */
        goto ERRRES;    /* goto error reset */
     }
    /* not end of frame so check for a transmit interrupt pending */
    if((c & 0x10) == 0)
        /* no transmit interrupt so end interrupt routine */
        goto ENDINT;
        }

TXINT:/********************** TRANSMIT DRIVER ***************************/
      /* After we have determined that we have received a transmit interrupt*/
      /* we check to see if we are at end of frame by checking its size. If */
      /* not, then we send out character pointed  at by frame pointer and   */
      /* then end the interrupt. If we are at end of frame, we clear frame  */
      /* counter and reset the transmit interrupt flag.                     */
      /**********************************************************************/

              if(fsize > 0)   /* are at the end of the frame ? */
        {
              fsize--;                        /* decrement frame size */
              outp(0x238,frame[fcnt++]);      /* NO - send character */
              goto ENDINT;                    /* end the interrupt routine */
        }
        outp(0x239,0);    /* must be at the end of the frame */
        fcnt = 0;         /* clear the frame count */
        outp(0x239,0x28); /* reset transmit interrupts */
        goto ENDINT;      /* end the interrupt routine */
        }
ERRRES:
   outp(0x239,0);         /* error reset */
   icnt = 0;              /* reuse the same buffer */
   outp(0x239,0x30);
ENDINT:
   outp(0x20,0x20);       /* End of interrupt report */
        return;
}

/******************************** MDATreq() ********************************
 *  MDATreq: Transmit LAPB output requests. When a buffer is ready to be     *
 *  transmitted, that buffer is sent to MDATreq routine where we check to see*
 *  if data is currently being transmitted. If it is, we return to calling   *
 *  routine and try again later. If no data is being transmitted, we check   *
 *  transmit buffer register of synchronous controller chip to see if it is  *
 *  empty so that when it is, we can send out first character of frame. When *
 *  we can send out a character the pointers and counters are setup and       *
 *  first character is sent. After first character is sent, buffer is removed*
 *  from outbound queue and a second character is sent to help with system   *
 *  timing. At this point, interrupt driver will take over and continue to   *
 *  send remaining data until last byte of frame is delivered along with     *
 *  frames CRC.                                                              *
 ***************************************************************************/

void MDATreq (msdu)
buf_type msdu;              /* Transmitt buffer pointer */
{
int rval, delay;           /* local varibles */
        if(fsize != 0)     /* if we are still transmitting a frame return */
                return;
        outp(0x239,0);
        fcnt=0;
        while (((rval = inp(0x239)) & 4) == 0);  /* test for buffer empty */

        frame = (char *)(BuffData(msdu));   /* get pointer to user buffer */
        fsize = (BuffSize(msdu)-1);         /* get size minus first char */
        QRemove(msdu);                      /* remove buffer from queue */
        delay=0;
        outp(0x238,frame[fcnt++]);          /* send first byte of frame */

        msdu=(buf_type)MDATcon(msdu,mac);   /* and confirm the buffer */
        while (((rval = inp(0x239)) & 4) == 0);  /* test for buffer empty */

        fsize--;                            /* decrement frame size */
        outp(0x238,frame[fcnt++]);          /* send second byte of frame */

        outp(0x239,0);
        delay = 0;
        outp(0x239,0xC0);                   /* process end of frame CRC */

        return;
```

End Listing Two

Listing Three

```
/***************************************************************************
 *  Poll timer queue for any expired timers. Next check to see if there is  *
 *  any X.25 traffic to be moved on Inbound or Outbound packet queue. The    *
 *  last process we check is if Baddr Array is low on preallocated receive   *
 *  buffers that are used by RCV2.C Interrupt Handler.                       *
 ***************************************************************************/
/* AUTHOR - Michael T. Thompson, Planning Systems Inc.  5/23/89  */

poldat()
{
        int i;  /* index varible */
        do_timer_queue();  /* Test for expaired timers */
        do_lapb_queue();   /* Check for X.25 Traffic */
        if(rbufcnt < 4)    /* replenish array if less than four pointers */
        {
                for(i=0;i<9;i++)        /* check all receive buffers */
                {
                   if(Baddr[i] == NULL) /* If Null, pointer has been used */
                   {    /* Get new buffer pointer & assign it to the array */
                    Baddr[i] = getbuf(&bpool,xbufsiz+30);
                    rbufcnt++;          /* Increment buffer count */
                   }
                }
        }
}       /* END OF POLDAT */
```

End Listings

Listing One *(Text begins on page 38.)*

```
/* Cheap Com  by Don Gaspar */
/* Requires MPW C and the Comm Toolbox. The complete program, including */
/* the resource file, header file, and make, are available electronically. */
/* These are the standard Mac includes for all managers */

#include <values.h>
#include <types.h>
#include <Resources.h>
#include <QuickDraw.h>
#include <fonts.h>
#include <events.h>
#include <windows.h>
#include <menus.h>
#include <textedit.h>
#include <dialogs.h>
#include <Controls.h>
#include <desk.h>
#include <toolutils.h>
#include <memory.h>
#include <Lists.h>
#include <SegLoad.h>
#include <Files.h>
#include <Packages.h>
#include <OSEvents.h>
#include <OSUtils.h>
#include <DiskInit.h>
#include <Traps.h>
#include <String.h>
#include <Strings.h>

#include <CRMIntf.h>  // Communications Resource Manager stuff
#include <CMIntf.h>   // Connection Manager stuff
#include <FTIntf.h>   // File Transfer Manager stuff
#include <TMIntf.h>   // Terminal Manager stuff
#include <CTBUtils.h> // Communications Toolbox Utility stuff

#include "CommTypes.h"   // communications types, etc.
#include "CheapComm.h"   // constants, forward declarations, etc.

/*  global variables */
Boolean      gHasWaitNextEvent; // does user's machine have WaitNextEvent?
Boolean      gInBackground;     // are we in the background?
Boolean      gStopped;          // are we stopped?

TermHandle   gTerm;           // handle to terminal record
ConnHandle   gConn;           // handle to connection record
FTHandle     gFT;             // handle to file transfer record

Ptr          gBuffer;        // global connection buffer
long         gFTSearchRefNum;
Boolean      gStartFT;        // are we doing a file transfer?
Boolean      gWasFT;          // was a file transfer in progress?

short        gDummy;
Handle       gCache;         // buffer for last terminal line received/sent
TEHandle     gTE;            // buffer for terminal emulator
ControlHandle gScrollHHandle, gScrollVHandle;

#pragma segment Main
/*      Sends data out via choosen connection */
pascal long     TermSendProc(thePtr, theSize, refCon, flags)
        Ptr     thePtr;
        long    theSize;
        long    refCon;
        short   flags;
{
        CMErr   theErr;
        long    termSendProc = 0L;
        if (gConn != nil) {
             theErr = CMWrite(gConn,thePtr,&theSize,cmData,false,nil,0,flags);
                if (theErr == noErr)
                        termSendProc = theSize;
        }
        return(termSendProc);
}
#pragma segment Main
/*  Gets the data from the connection tool and sends it to the terminal tool */
pascal void TermRecvProc()
{
        CMErr           theErr;
        CMStatFlags     status;
        CMBufferSizes   sizes;
        short           flags;
        if (gConn != nil && gTerm != nil) {
                theErr = CMStatus(gConn, sizes, &status);
                if (theErr == noErr) {
if ((status & (cmStatusOpen+cmStatusDataAvail)) != 0 && sizes[cmDataIn] != 0) {
                                if (sizes[cmDataIn] > kBufferSize)
                                        sizes[cmDataIn] = kBufferSize;
theErr=CMRead(gConn, gBuffer, &sizes[cmDataIn], cmData, false, nil, 0, &flags);
        if (theErr  == noErr)   // give it to terminal emulation buffer
            sizes[cmDataIn] = TMStream(gTerm, gBuffer, sizes[cmDataIn], flags);
           }
        }
        else
        ;                       // Connection Manager will handle this
    }
}
#pragma segment Main
/* Gets the connection environments for FT or Term tool */
pascal OSErr    ToolGetConnEnvirons(refCon, theEnvirons)
        long    refCon;
        ConnEnvironRec  *theEnvirons;
{
        OSErr   toolGetConnEnvirons = envNotPresent;
        if(gConn != nil)
                toolGetConnEnvirons = CMGetConnEnvirons(gConn,theEnvirons);
        return(toolGetConnEnvirons);
}
```

```
#pragma segment Main
/* Sends data during a file transfer */
pascal long     FTSendProc(thePtr, theSize, refCon, channel, flags)
        Ptr         thePtr;
        long        theSize;
        long        refCon;
        CMChannel   channel;
        short       flags;
{
        CMErr   theErr;
        long    ftSendProc = 0L;
    if (gConn != nil) {
      theErr = CMWrite(gConn, thePtr,& theSize, channel, false, nil, 0, flags);
                if(theErr == noErr)
                        ftSendProc = theSize;
        }
        return(ftSendProc);
}
#pragma segment Main
/* Gets the data during a data transfer */
pascal long     FTReceiveProc(thePtr, theSize, refCon, channel, flags)
        Ptr         thePtr;
        long        theSize;
        long        refCon;
        CMChannel   channel;
        short       *flags;
{
        CMErr       theErr;
        long        ftReceiveProc = 0L;
     if (gConn != nil) {
       theErr = CMRead(gConn, thePtr, &theSize, channel, false, nil, 0, flags);
                if (theErr == noErr)
                        ftReceiveProc = theSize;
        }
        return(ftReceiveProc);
}
#pragma segment Main
/* Sets file transfer flag if an autoreceive string was found */
pascal void AutoRecCallBack(theConn, data, refNum)
        ConnHandle      theConn;
        Ptr             data;
        long            refNum;
{
        if (gFTSearchRefNum == refNum)
                gStartFT = true;
}
#pragma segment Main
/* Checks if file transfer has autoreceive string; adds a search to find it */
void AddFTSearch()
{
        Str255  tempStr;
        if (gFT != nil && gConn != nil) {
                //tempStr = (*gFT)->autoRec;
        if ((*gFT)->autoRec[0] != 0) {
         gFTSearchRefNum = CMAddSearch(gConn,(*gFT)->autoRec, cmSearchSevenBit,
                                (ProcPtr)AutoRecCallBack);
                        if (gFTSearchRefNum == -1) {
                             AlertUser("Couldn't add stream search\0", false);
                                gFTSearchRefNum = 0;
                        }
                }
        }
}
#pragma segment Main
/* Initiates a file transfer send from menu command */
void DoSend()
{
        SFReply         theReply;
        Point           where;
        short           numTypes;
        SFTypeList      typeList;
        FTErr           anyErr;
        if(gFT != nil) {
                SetPt(&where,100,100);

                if(((**gFT).attributes & ftTextOnly) != 0) {
                        typeList[0] = 'TEXT';
                        numTypes = 1;
                }
                else
                        numTypes = -1;
    sfgetfile(&where, "File to send", nil, numTypes, typeList, nil, &theReply);
                if(theReply.good) {
                        anyErr = FTStart(gFT, ftTransmitting, &theReply);
                        if(anyErr != noErr)
                                ; // file transfer tool will alert user
                }
        }
}
#pragma segment Main
/* Initiates a file transfer receive from menu */
void DoReceive()
{
        SFReply   theReply;
        OSErr     anyErr;

        if (gFT != nil) {
                theReply.vRefNum = 0;
                //theReply.fName = '';
                gStartFT = false;
                if (gConn != nil ) {
                        if ((**gFT).autoRec != "\0" && gFTSearchRefNum != 0) {
                                CMRemoveSearch(gConn, gFTSearchRefNum);
                                gFTSearchRefNum = 0;
                        }
                }
                anyErr = FTStart(gFT, ftReceiving, &theReply);
                if(anyErr != noErr)
                        ; // file transfer tool will alert user
        }
}
```

(continued on page 96)

Listing One *(Listing continued, text begins on page 38.)*

```
#pragma segment Main
/* Initiates a connection */
void OpenConnection()
{
        CMErr           theErr;
        CMBufferSizes   sizes;
        CMStatFlags     status;
        if(gConn != nil) {
                theErr = CMStatus(gConn, sizes, &status);
                if(theErr == noErr)
                        if((status & (cmStatusOpen + cmStatusOpening)) == 0)
                                theErr =CMOpen(gConn, false, nil, -1);
                        if (theErr != noErr)
                            ; // connection tool will tell user if there's an errror
        }
}
#pragma segment Main
/* Cancels connection */
void CloseConnection()
{
        CMErr           theErr;
        CMBufferSizes   sizes;
        CMStatFlags     status;
        if (gConn != nil) {
                theErr = CMStatus(gConn, sizes, &status);
                if(theErr == noErr)
                        if ((status & (cmStatusOpen+cmStatusOpening)) != 0)
                                theErr = CMClose(gConn, false, nil ,0, true);
                if (theErr != noErr)
                        ; // connection tool will handle error
        }
}
#pragma segment Main
/* tries to get default tool proc ID, otherwise gets first one it can find */
short   FindToolID(toolClass)
        OSType  toolClass;
{
        Str255  toolName;
        OSErr   anyErr;
        short   procID = -1;
        if (toolClass == classTM) {
                StuffHex(&toolName,kDefaultTermTool);
                procID = TMGetProcID(toolName);
                if(procID == -1) {
                        anyErr = CRMGetIndToolName(toolClass,1, toolName);
                        if (anyErr == noErr)
                                procID = TMGetProcID(toolName);
                }
        }
        else if (toolClass == classCM) {
                StuffHex(&toolName,kDefaultConnTool);
                procID = CMGetProcID(toolName);
                if(procID == -1) {
                        anyErr = CRMGetIndToolName(toolClass,1, toolName);
                        if (anyErr == noErr)
                                procID = CMGetProcID(toolName);
                }
        }
        else if (toolClass == classFT) {
                StuffHex(&toolName,kDefaultFTTool);
                procID = FTGetProcID(toolName);
                if(procID == -1) {
                        anyErr = CRMGetIndToolName(toolClass,1, toolName);
                        if (anyErr == noErr)
                                procID = FTGetProcID(toolName);
                }
        }
        return(procID);
}
#pragma segment Main
/* this is click loop for terminal emulation to track */
pascal Boolean clikLoop(refcon)
        long    refcon;
{
        return(true);
}
#pragma segment Initialize
/* this function sets up CommToolbox for what we need;
        it should resemble most other Macinotosh toolbox calls  */
void InitCommTB()
{
        (void)InitCTBUtilities();       //      Comm Toolbox Utilities
        (void)InitCRM();                //      Communications Resource Manager
        //      initialize the Terminal Manager
        if (InitTM() == tmNoTools)                      //      Did we fail?
                AlertUser("No terminal tools found\0", true);
        //      Initialize the Connection Manager
        if(InitCM()== cmNoTools)                        //      failure?
                AlertUser("No connection tools found\0", true);
        //      Initialize the File Transfer Manager
        if(InitFT() == ftNoTools)                               // failure?
                AlertUser("No file transfer tools found\0",false);
        gTerm = nil;                    // initialize our globals
        gConn = nil;
        gFT = nil;
        gCache = nil;
        gFTSearchRefNum = 0;
}
#pragma segment Main
/*      this will cache all data coming in through serial port */
pascal long cacheProc(refCon, theTermData)
        long            refCon;
        TermDataBlock   *theTermData;
{
        long            sizeCached;
        TermEnvironRec  theEnvirons;
        theEnvirons.version = curTermEnvRecVers;
        theEnvirons.termType = tmTextTerminal;
```

(continued on page 98)

Listing One *(Listing continued, text begins on page 38.)*

```
        (void)TMGetTermEnvirons(gTerm, &theEnvirons);
        if (theTermData->theData == nil)
                return(-1);
        if(gCache != nil)       //      is it valid?
                        DisposHandle(gCache);
        HLock((Handle)theTermData->theData);
        gCache = theTermData->theData;
        if(HandToHand(&gCache)) {
                        DisposHandle(gCache);
                        sizeCached = -1;
                }
                else {
                        sizeCached = GetHandleSize(gCache);
                }
        HUnlock((Handle)theTermData->theData);
        if(theTermData->flags == tmTextTerminal && sizeCached >0L) {
                /*HandAndHand(gCache, (**gTE).hText);
                (**gTE).teLength += 80;
                (**gTE).nLines += 1;*/
                ((Ptr)*gCache,80L,gTE);
                //(**gTE).viewRect.top -= (**gTE).lineHeight;
                //(**gTE).destRect.top -= (**gTE).lineHeight;
                //TECalText(gTE);
                //TEScroll(0,-(**gTE).lineHeight,gTE);
        }
        return(tmNoErr);
}
#pragma segment Main
/* gets window and create session */
Boolean DoNewWindow()
{
        WindowPtr       window;
        Rect            theRect;
        short           procID;
        CMBufferSizes   sizes;
        Rect            tempRect;
        short           index;
        Rect            r;
        window = GetNewWindow(rWindow, nil, (WindowPtr)-1);
        SetPort(window);
        /* no cache, breakproc, or clikloop */
        gTerm = TMNew(&theRect,&theRect, tmSaveBeforeClear + tmAutoScroll,
                 procID, window, (ProcPtr)TermSendProc,(ProcPtr)cacheProc,nil,
                /*(ProcPtr)clikLoop*/nil, (ProcPtr)ToolGetConnEnvirons,0,0);
        SetRect(&r,theRect.left,-theRect.bottom,theRect.right,theRect.top);
        gTE = TENew(&r,&r);
   (**gTE).txSize = 9;
   (**gTE).txFont = monaco;
   (**gTE).viewRect.bottom = (((**gTE).viewRect.bottom - (**gTE).viewRect.top)/
   (**gTE).lineHeight)*(**gTE).lineHeight + (**gTE).viewRect.top;
        TEAutoView(true,gTE);
   /* custom configure with personal settings -- store as a file later */
        (void)TMSetConfig(gTerm, "Scroll Smooth\0");
        if(gTerm == nil)
                AlertUser("Can't create a terminal tool\0", true);
        HLock((Handle)gTerm);
        /* connection tool */
        procID = FindToolID(classCM);
        if(procID == -1)
                AlertUser("No connection tools found/0", true);
      sizes[cmDataIn] = kBufferSize*10;  // data channel; large incoming buffer
      sizes[cmDataOut] = kBufferSize;
      sizes[cmCntlIn] = 0;
      sizes[cmCntlOut] = 0;
      sizes[cmAttnIn] = 0;
      sizes[cmAttnOut] = 0;
        gConn = CMNew(procID, cmData, sizes, 0,0);
        (void)CMSetConfig(gConn,"Baud 9600, Bits 7, StopBits 1, Parity Even,
                    ModemType Other, PhoneNumber \0429,1800-346-0145\042\0");
        if(gConn == nil)
                AlertUser("Can't create a connection tool/0", true);
        HLock((Handle)gConn);
/* allocate space for reads/writes using number returned by connection tool */
        gBuffer = NewPtrClear(sizes[cmDataIn]);
        if(MemError() != noErr)
                AlertUser("Out of memory\0", true);
        /* file transfer tool */
        procID = FindToolID(classFT);
        if(procID == -1)
                AlertUser("No file transfer tools found\0", false);
        /* no read/write proc -- tool has its own */
        gFT = FTNew(procID, 0 ,(ProcPtr)FTSendProc, (ProcPtr)FTReceiveProc,
                nil, nil, (ProcPtr)ToolGetConnEnvirons, window, 0L,0L);
        if(gFT == nil)
                AlertUser("Can't create a file transfer tool\0", true);
        HLock((Handle)gFT);
        gWasFT = false;
        gStartFT = false;
        gFTSearchRefNum = 0;
        AddFTSearch();
        return(true);
}
#pragma segment Main
/* Updates the window */
void DoUpdate(window)
        WindowPtr       window;
{
        RgnHandle       savedClip;
        GrafPtr         savedPort;

        if (IsAppWindow(window)) {
                GetPort(&savedPort);
                SetPort(window);
                /* clip to the window content */
                savedClip = NewRgn();
                GetClip(savedClip);
                ClipRect(&window->portRect);
```

```
                DrawControls(window);
                DrawGrowIcon(window);
                BeginUpdate(window);
                        if(gTerm != nil )
                                TMUpdate(gTerm, window->visRgn);
                        if(gTE != nil)
                                TEUpdate(&window->portRect,gTE);
                EndUpdate(window);
                SetClip(savedClip);
                DisposeRgn(savedClip);
                SetPort(savedPort);
        }
}
#pragma segment Main
/* suspends/resumes terminal window */
void DoResume(becomingActive)
        Boolean becomingActive;
{
        WindowPtr       theWindow;
        GrafPtr         savedPort;
        GetPort(&savedPort);
        theWindow = FrontWindow();
        while (theWindow!= nil) {
                if (IsAppWindow(theWindow)) {
                        SetPort(theWindow);
                        if(gTerm != nil)
                                TMResume(gTerm, becomingActive);
                        if(gConn != nil)
                                CMResume(gConn, becomingActive);
                        if(gFT != nil)
                                FTResume(gFT, becomingActive);
                }
                theWindow = (WindowPtr)(((WindowPeek) theWindow)->nextWindow);
        }
        SetPort(savedPort);
}
#pragma segment Main
/* (de)activates window */
void DoActivate(window, becomingActive)
        WindowPtr       window;
        Boolean                 becomingActive;
{
        if (IsAppWindow(window)) {      // does window belong to us?
                SetPort(window);        // set current port
                if(gConn != nil)        // do we have a valid connection?
                        CMActivate(gConn, becomingActive);// activate it
                if(gTerm != nil)        // do we have a terminal?
                        TMActivate(gTerm, becomingActive); // activate it
                if(gFT != nil)          // do we have a valid file transfer?
                        FTActivate(gFT, becomingActive);// activate it
        }
}
#pragma segment Main
/* tries to pass event to a tool if window is a tool window;
                                        handles event if appropriate */
Boolean DoToolEvent(event, window)
        EventRecord     *event;
        WindowPtr       window;
{
        Boolean         doToolEvent;
        if (window != nil && !IsAppWindow(window)) { // is window valid?
                doToolEvent = true;
                /* copies of commtb record must be in refCon field of
                        window for changing the settings        */
                if(gFT != nil && gFT == (FTHandle)GetWRefCon(window))
                        FTEvent(gFT,event); // handle file transfer manager event
                else if(gConn != nil && gConn == (ConnHandle)GetWRefCon(window))
                        CMEvent(gConn,event); // handle connection manager event
                else if(gTerm != nil && gTerm == (TermHandle)GetWRefCon(window))
                        TMEvent(gTerm,event); // handle terminal manager event
                else
                        doToolEvent = false;
        }
        else
                doToolEvent = false;
        return(doToolEvent);
}
#pragma segment Main
/* idles all communications tools; this was taken from the Surfer pascal
        example provided from Apple -- you can get it from APDA.*/
void DoIdle()
{
        WindowPtr   theWindow;
        Boolean         doFT, doTM;
        GrafPtr         savedPort;
        GetPort(&savedPort);
        theWindow = FrontWindow();
        while (theWindow != nil) {
                if (IsAppWindow(theWindow)) {
                        SetPort(theWindow);
                        //TEIdle(gTE);
                        if(gConn != nil)
                                CMIdle(gConn);
                        doFT = false;
                        doTM = true;
                        if (gFT != nil ) {
                                if (((**gFT).flags  & ftIsFTMode) != 0) {
                                        doFT = true;
                                        gWasFT = true;
                                  if(((**gFT).attributes & ftSameCircuit) != 0)
                                                doTM = false;
                                }
                                else {
                                        if (gWasFT) {
                                                gWasFT = false;
                                              if(((**gFT).flags & ftSucc) == 0)
                                                        ;
                                                AddFTSearch();
                                        }
                                        if(gStartFT)
                                                DoReceive();
                                        }
                                        if (doFT)
                                                FTExec(gFT);
                                } /* if gFT != nil  */
                                if(gTerm != nil) {
                                        if (doTM) {
                                                TMIdle(gTerm);
                                                TermRecvProc();
                                        }
                                }/* gTerm != nil */
                        }
                        theWindow = (WindowPtr)(((WindowPeek)theWindow)->nextWindow);
                }
                SetPort(savedPort);
}
```

End Listing

Listing One *(Text begins on page 54.)*

```
/*************************************************************************
 * SS.C -- Sample Sorted Sequential Suffix Search (c) 1989 Walter Williams
 *************************************************************************/

#include <stdio.h>
#include <string.h>
#include <malloc.h>

#ifndef TRUE
#define TRUE    1
#define FALSE   0
#endif

typedef struct snode_S
    {
    struct snode_S *prev;   /* Address of previous node in list     */
    struct snode_S *next;   /* Address of next node in list         */
    unsigned int pfxlen;    /* Number of characters in prefix       */
    char key[1];            /* First character of key               */
    } snode_T, *snode_TP;

/********************** Function Prototypes ***************************/
snode_TP Search(char *, snode_TP, int *, unsigned int *);
snode_TP Insert(char *, snode_TP);
snode_TP Delete(char *, snode_TP);

/*-----------------------------------------------------------------------*/
/* SEARCH() -- Search the list for a pattern key.                        */
/* 'pattern' is a null terminated string containing the key which is     */
/*           the object of the search.                                   */
/* 'list' is the address of a dummy node which contains head and tail    */
/*           pointers for a linked list.                                 */
/* 'exact' is the address of a flag which is TRUE for an exact match     */
/*           and FALSE if the pattern is not found.                      */
/* 'match' is the address of an unsigned int to use as a match counter   */
/* The return value is a pointer to the structure containing the         */
/* matching key, or the next largest node if the pattern was not found.  */
/*-----------------------------------------------------------------------*/

snode_TP Search(char *pattern, snode_TP list, int *exact, unsigned int *match)
    {
    snode_TP cnode;     /* Pointer to current node              */
    char *sp;           /* Suffix pointer                       */
    int tm= 0;          /* Temp storage for match count         */
    /***/
    *exact= FALSE;                      /* Assume unsuccessful search   */
    *match= tm;
    for (cnode= list->next;  cnode != list;  cnode= cnode->next)
        {
                                /* Compare match count to prefix count  */
        if (tm < cnode->pfxlen)
            continue;
        else if (tm > cnode->pfxlen)
            break;
        else /* (tm == cnode->pfxcnt) */
            {
                /* Compare the actual key suffix, maintain match count */
            sp= cnode->key + cnode->pfxlen;
            while (*pattern == *sp && *sp && *pattern)
                {
                ++sp;
                ++pattern;
                ++tm;
                }
                    /* Done if suffix greater than or equal to pattern  */
            if (*pattern < *sp )
                {
                break;
                }
            else if (*pattern == '\0' && *sp == '\0')
                {
                *match= tm;
                *exact= TRUE;
                break;
                }
            }
        *match= tm;
        }
    return (cnode);
    }

/*--- INSERT() Adds an item to the list.  ---*/
snode_TP Insert(char *pattern, snode_TP list)
    {
    snode_TP cnode;         /* Node we are inserting                */
    snode_TP nnode;         /* Next node after cnode                */
    char *sp;               /* Pointer to suffix                    */
    unsigned int match;
    int exact;
    /***/
                                /* Find spot where we insert the node   */
    nnode = Search(pattern, list, &exact, &match);
    if (exact == TRUE)          /* Skip to first non-matching key       */
        {
        nnode = nnode->next;
        while (nnode != list && nnode->key[nnode->pfxlen] == '\0')
            nnode = nnode->next;
        }
                                /* Allocate space for the new node      */
    cnode = (snode_TP) malloc(sizeof(snode_T) + strlen(pattern));
    cnode->pfxlen = match;
    strcpy(cnode->key, pattern);
                                /* Link it into the list ahead of nnode */
    cnode->next = nnode;
    cnode->prev = nnode->prev;
    nnode->prev->next = cnode;
    nnode->prev = cnode;
                                /* Update pfxlen in following node      */
    sp = nnode->key + nnode->pfxlen;
    if (cnode->pfxlen == nnode->pfxlen)
        {                       /* Compare the two suffixes             */
        nnode->pfxlen= 0;
        while (*sp == *pattern && *pattern && *sp)
            {
            ++sp;
            ++pattern;
            ++nnode->pfxlen;
            }
        }
    return (cnode);
    }

/*--- DELETE() Deletes an item from the list ---*/
snode_TP Delete(char *pattern, snode_TP list)
    {
    snode_TP cnode;         /* Node we are deleting                 */
    snode_TP nnode;         /* Next node after cnode                */
    int exact;              /* Flag set if exact match              */
    unsigned int match;     /* No. of characters matched in pattern */
    /***/
                            /* Find the node we want to delete      */
    cnode = Search(pattern, list, &exact, &match);
    if (exact == FALSE)     /* Abort if not an exact match          */
        {
        printf("%s not found\n", pattern);
        nnode= NULL;
        }
    else
        {                   /* Remove it from the list              */
        cnode->next->prev = cnode->prev;
        cnode->prev->next = cnode->next;
        nnode = cnode->next;/* Save for return value                */
                            /* Update suffix in following node      */
        if (cnode->pfxlen < cnode->next->pfxlen)
            cnode->next->pfxlen = cnode->pfxlen;
                            /* Release deleted node                 */
        free((char *) cnode);
        printf("%s deleted\n", pattern);
        }
    return (nnode);
    }
```

End Listing

Listing One *(Text begins on page 64.)*

```
// Line Editor example using Zinc Interface Library -- (c) Gary Entsminger

#include <ui_win.hpp>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// Derive a new class which adds File I/O to a window.
class Line_editor: public UIW_WINDOW
{
public :
   Line_editor();       // Constructor
  ~Line_editor();       // Destructor
   char * line_buffer[64];
 };  // end class Line_editor declaration

// Define Line_editor's constructor
Line_editor :: Line_editor() : UIW_WINDOW(2,2,70,5,
                    WOF_NO_FLAGS, WOAF_NO_FLAGS)
{
FILE *textfile; // pointer to file
char ln[64];    // local char array to hold lines read from file

   // Open file
   if ((textfile = fopen("mk.bat", "r")) == NULL)
   {
      printf("Error opening text file\n Aborting.");
      exit(0);
   }

   // Read first 64 characters of file
   fseek(textfile,SEEK_SET,0);
   fread(ln,64,1,textfile);

   // Close file
   fclose(textfile);

  strcpy(*line_buffer,ln);
}  // end Line_editor constructor

  //  Define Line_editor's destructor.
Line_editor:: ~Line_editor()
{
FILE *textfile; // pointer to file
char ln[64];  // char array to hold a line read from file

  // Open Make file (MK.BAT)
   if ((textfile = fopen("mk.bat", "w")) == NULL)
   {
      printf("Error opening text file. Aborting. \n");
      exit(0);
   }

// Write edited text to the file
strcpy(ln,*line_buffer);
fwrite(ln,64,1,textfile);

   // Close file
 fclose(textfile);

} // end destructor.

main()
{
// Construct the display, trying for graphics first.
UI_DISPLAY *display = new UI_DOS_BGI_DISPLAY;
if (!display->installed)
{
delete display;
display = new UI_DOS_TEXT_DISPLAY;
}

// Construct the event manager and add three devices to it.
UI_EVENT_MANAGER *eventManager =
  new UI_EVENT_MANAGER(100, display);

*eventManager
                  + new UI_BIOS_KEYBOARD
                  + new UI_MS_MOUSE
                  + new UI_CURSOR;

// Construct the window manager.
UI_WINDOW_MANAGER *windowManager =
   new UI_WINDOW_MANAGER(display, eventManager);

// Construct a new Line_editor in window 1.
Line_editor *window1 = new Line_editor();

// Construct a new string field for the Line_editor in window1.
UIW_STRING *stringfield = new UIW_STRING(1,1,64,
            *window1->line_buffer, 64,
            STF_NO_FLAGS, WOF_NO_ALLOCATE_DATA);

// Add the window objects to the line_editor in window 1.
*window1
+ new UIW_BORDER
+ new UIW_MAXIMIZE_BUTTON
+ new UIW_MINIMIZE_BUTTON
+ new UIW_SYSTEM_BUTTON
+ new UIW_TITLE("MAKE .BAT editor", WOF_JUSTIFY_CENTER)
+ stringfield;  // Add the string field we constructed to window1

// Add  window1 to the window manager.
*windowManager + window1;
```

(continued on page 102)

Listing One *(Listing continued, text begins on page 64.)*

```
// Loop for user response.
int ccode;
UI_EVENT event;

do
{
// Get input from the user.
eventManager->Get(event, Q_NORMAL);

// Send event information to the window manager.
ccode = windowManager->Event(event);
}
while (ccode != L_EXIT && ccode != S_NO_OBJECT);

// Manually decouple stringfield from window1. Destruct all the objects we
// constructed in the opposite order in which we created them.
*window1 - stringfield;
delete stringfield;
delete window1;
delete windowManager;
delete eventManager;
delete display;
} // end main.
```

End Listing One

Listing Two

```
### Production
OPTIONS=-O -w -P

### Debugging
#OPTIONS=-v -w -P

### Zinc information
INCLUDE=..\include
LIB=..\lib
!if !$d(MODEL)
MODEL=1
!endif

.SWAP
.PATH.obj=$(MODEL)
.PATH.lib=$(LIB)
.cpp.exe :
        tcc $(OPTIONS) -m$(MODEL) -I$(INCLUDE) -L$(LIB)
 {$< zil$(MODEL).lib graphics.lib
```

End Listing Two

Listing Three

```
#include <ui_win.hpp>
#include "editor.hlh"

main()
{
// Initialize the display, try for graphics first.
UI_DISPLAY *display = new UI_DOS_BGI_DISPLAY;
if (!display->installed)
{
   delete display;
   display = new UI_DOS_TEXT_DISPLAY;
}

// Initialize the event manager.
UI_EVENT_MANAGER *eventManager = new UI_EVENT_MANAGER(100, display);
*eventManager + new UI_BIOS_KEYBOARD + new UI_MS_MOUSE + new
UI_CURSOR;

// Initialize the window manager.
UI_WINDOW_MANAGER *windowManager =
   new UI_WINDOW_MANAGER(display, eventManager);

// Initialize the help window system.
_helpSystem = new UI_HELP_WINDOW_SYSTEM("notepad.hlp",
   windowManager, HELP_GENERAL);

// Initialize the error window system.
_errorSystem = new UI_ERROR_WINDOW_SYSTEM;

// Create a field editor window.
UIW_WINDOW *editor = new UIW_WINDOW(4, 5, 66, 12,
              WOF_NO_FLAGS, WOAF_NO_FLAGS);

// Add window objects to editor.
*editor
  + new UIW_BORDER
  + new UIW_MAXIMIZE_BUTTON
  + new UIW_MINIMIZE_BUTTON
  + new UIW_SYSTEM_BUTTON
  + new UIW_TITLE("FieldEditor", WOF_JUSTIFY_CENTER)

  + new UIW_PROMPT(2, 1, "To:", WOF_NO_FLAGS)
  + new UIW_STRING(6, 1, 15, "Dr. Dobbs folk", 40, STF_NO_FLAGS,
              WOF_BORDER)

  + new UIW_PROMPT(28, 1, "Date:", WOF_NO_FLAGS)
  + new UIW_DATE(35, 1, 20, &UI_DATE(), "",
                                DTF_SYSTEM | DTF_ALPHA_MONTH, WOF_BORDER,
                NO_HELP_CONTEXT)

  + new UIW_PROMPT(2, 2, "Message:", WOF_NO_FLAGS)
  + new UIW_TEXT(2, 3, 60, 4, "", 1028, TXF_NO_FLAGS, WOF_BORDER);

// Add field editor to the window manager.
*windowManager + editor;

// Wait for user response.
int ccode;
UI_EVENT event;
do
{
              eventManager->Get(event, Q_NORMAL);
              ccode = windowManager->Event(event);
} while (ccode != L_EXIT && ccode != S_NO_OBJECT);

// Clean up.
  delete _helpSystem;
  delete _errorSystem;
  delete windowManager;
  delete eventManager;
  delete display;
}

// Contents of editor.hlh........
// This file was created by the GENHELP utility.

const int HELP_GENERAL = 1;   // General Help
const int HELP_EDITOR  = 2;   // Editor  Help
```

End Listings

Listing One *(Text begins on page 72.)*

```
/* You will need the Paradox Engine to compile this program. If you have   */
/* placed directory for Engine in your INCLUDE path and LIBRARY path, the  */
/* modules should compile fine with: tcc -ml mci parse send pxengtcl.lib   */
/* Otherwise, specify where Engine header file and library files are       */
/* using the -I and -L compiler switches.                                  */

#define MCIMAIN

#include <stdio.h>
#include <stdlib.h>
#include <pxengine.h>
#include "mci.h"

void closeTables(void);
void getFldNames(void);
void makeTblExist(char *tblName,int nFlds,char **fld,char **types,
                  int
keyFields);
void openTables(void);
void shutdownEngine(void);
void startEngine(void);

/**** DEFINE HEADER TABLE INFORMATION ****/
char *headerFields[] =
{
  "Date Received","Message#","Subject","From User","From ID"
};
char *headerTypes[] =
{
  "D","N","A40","A20","A8"
};
int headerKeyFields = 2;
#define headerNFields sizeof(headerTypes)/sizeof(char*)

/**** DEFINE MESSAGE TABLE INFORMATION ****/
char *msgFields[] =
{
  "Message#","Line#","Text"
};
char *msgTypes[] =
{
  "N","N","A80"
};
int msgKeyFields = 2;
#define msgNFields sizeof(msgTypes)/sizeof(char*)

/**** DEFINE ROUTE TABLE INFORMATION ****/
char *routeFields[] =
{
  "Message#","TOorCC","User Name","User ID"
};
char *routeTypes[] =
{
  "N","A5","A20","A8"
};
int routeKeyFields = 3;
#define routeNFields sizeof(routeTypes)/sizeof(char*)

int lastPXErr;
FIELDHANDLE fh[] = {1};
TABLEHANDLE headerTbl,msgTbl,routeTbl;

int main(int argc,char *argv[])
{
        startEngine();
        openTables();
        getFldNames();
        if (argc > 1)
             parse(argv[1]);    /* only add to tables if filename passed in */
        send();
        closeTables();
        shutdownEngine();
        return 0;
}

void startEngine(void)
{
        PXCheck(0x101,PXInit());
}

void shutdownEngine(void)
{
        PXCheck(0x110,PXExit());
}

void openTables(void)
{

makeTblExist("Header",headerNFields,headerFields,headerTypes,headerKeyFields);
makeTblExist("Message",msgNFields,msgFields,msgTypes,msgKeyFields);
makeTblExist("Route",routeNFields,routeFields,routeTypes,routeKeyFields);
       PXCheck(0x120,PXTblOpen("Header",&headerTbl,0,0));
       PXCheck(0x121,PXTblOpen("Message",&msgTbl,0,0));
       PXCheck(0x122,PXTblOpen("Route",&routeTbl,0,0));
       PXCheck(0x123,PXRecBufOpen(headerTbl,&headerRec));
       PXCheck(0x125,PXRecBufOpen(msgTbl,&msgRec));
       PXCheck(0x126,PXRecBufOpen(routeTbl,&routeRec));
}
void closeTables(void)
{
        PXCheck(0x130,PXTblClose(headerTbl));
        PXCheck(0x131,PXTblClose(msgTbl));
        PXCheck(0x132,PXTblClose(routeTbl));
}

void getFldNames(void)
{
        PXCheck(0x140,PXFldHandle(headerTbl,"Date Received",&hdrDateFld));
        PXCheck(0x141,PXFldHandle(headerTbl,"Message#",&hdrMsgFld));
        PXCheck(0x142,PXFldHandle(headerTbl,"Subject",&hdrSubjectFld));
        PXCheck(0x143,PXFldHandle(headerTbl,"From User",&hdrUserFld));
        PXCheck(0x144,PXFldHandle(headerTbl,"From ID",&hdrIDFld));
        PXCheck(0x145,PXFldHandle(msgTbl,"Message#",&msgMsgNumFld));
        PXCheck(0x146,PXFldHandle(msgTbl,"Line#",&msgLineFld));
        PXCheck(0x147,PXFldHandle(msgTbl,"Text",&msgTextFld));
        PXCheck(0x148,PXFldHandle(routeTbl,"Message#",&routeMsgFld));
        PXCheck(0x149,PXFldHandle(routeTbl,"ToOrCC",&routeToFld));
        PXCheck(0x14A,PXFldHandle(routeTbl,"User Name",&routeUserFld));
        PXCheck(0x14B,PXFldHandle(routeTbl,"User ID",&routeIDFld));
        return;
}

void makeTblExist(char *tblName,int nFlds,char **fld,char **types,
                                                        int keyFields)
{
        int exist;
        PXCheck(0x150,PXTblExist(tblName,&exist));
        if (!exist)
        {
                PXCheck(0x151,PXTblCreate(tblName,nFlds,fld,types));
                if (keyFields)
                        PXCheck(0x152,PXKeyAdd(tblName,keyFields,fh,PRIMARY));
        }
        return;
}

void PXCheck(int loc,int errCode)
{
        lastPXErr=errCode;
        if(errCode==PXSUCCESS)
                return;
        switch (loc)
        {
                case 0x234:
                case 0x237:
                case 0x247:
                case 0x253:
                case 0x254:     return;
                default:  printf("Error '%s' at %04x\n",PXErrMsg(errCode),loc);
                                        exit (1);
        }
}

int PXLastErr(void)
{
        return lastPXErr;
}
```

End Listing One

Listing Two

```
#ifdef MCIMAIN
#define extern
#endif

#define MESSAGELENGTH 90
#define STARTOFNEWMESSAGE "Date:"
#define ENDOFSCRIPT "Command:"

int PXLastErr(void);
void PXCheck(int loc,int errCode);
void send(void);
void parse(char *filename);

extern short MessageID;
extern TABLEHANDLE headerTbl,msgTbl,routeTbl;
extern RECORDHANDLE headerRec,msgRec,routeRec;

extern FIELDHANDLE hdrDateFld;
extern FIELDHANDLE hdrMsgFld;
extern FIELDHANDLE hdrSubjectFld;
extern FIELDHANDLE hdrUserFld;
extern FIELDHANDLE hdrIDFld;

extern FIELDHANDLE msgMsgNumFld;
extern FIELDHANDLE msgLineFld;
extern FIELDHANDLE msgTextFld;

extern FIELDHANDLE routeMsgFld;
extern FIELDHANDLE routeToFld;
extern FIELDHANDLE routeUserFld;
extern FIELDHANDLE routeIDFld;
```

End Listing Two

Listing Three

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pxengine.h>
#include "mci.h"

void getCC(FILE *inMsgFp,char inBuffer[],int msgID);
void getHeaderInfo(FILE *inMsgFp,char inBuffer[],int msgID);
short getNextMsgID(short *MessageID);
void getTo(FILE *inMsgFp,char inBuffer[],int msgID);
void parseAddr(char inBuffer[],char Person[],char ID[]);
void parseDate(char *dateStr);
void whoFrom(FILE *inMsgFp,char inBuffer[],int msgID);
int convertMonth(char *str);

void parse(char *filename)
{
        FILE *inMsgFp;
```

(continued on page 106)

Listing Three *(Listing continued, text begins on page 72.)*

```
        char inBuffer[MESSAGELENGTH+1];
        short LineNumber;

        if ((inMsgFp = fopen(filename,"r")) == NULL)
        {
                perror(filename);
                PXExit();
                return;
        }

        /* Parse through text file searching for first MCI message. */
        while (fgets(inBuffer,MESSAGELENGTH,inMsgFp))
                if(strstr(inBuffer,STARTOFNEWMESSAGE) == inBuffer)
                        break;
        do
        {
        /* If STARTOFNEWMESSAGE is at beginning of current line, save last  */
        /* message and get remainder of header information for the message. */
        /* Otherwise, keep getting text.                                    */
                if (strstr(inBuffer,ENDOFSCRIPT) == inBuffer)
                        break;                  /* Reached end of messages */
                if (strstr(inBuffer,STARTOFNEWMESSAGE) == inBuffer)
                {
                        LineNumber = 1;
                        getNextMsgID(&MessageID);
                        getHeaderInfo(inMsgFp,inBuffer,MessageID);
                }
             if (strlen(inBuffer) > 0 && inBuffer[strlen(inBuffer)-1] == '\n')
                        inBuffer[strlen(inBuffer)-1] = '\0';
             PXCheck(0x300,PXPutAlpha(msgRec,msgTextFld,inBuffer));
             PXCheck(0x301,PXPutShort(msgRec,msgLineFld,LineNumber++));
             PXCheck(0x302,PXPutShort(msgRec,msgMsgNumFld,MessageID));
             PXCheck(0x303,PXRecAppend(msgTbl,msgRec));
        }while (fgets(inBuffer,MESSAGELENGTH,inMsgFp));
        fclose(inMsgFp);
        return;
}

void getHeaderInfo(FILE *inMsgFp,char inBuffer[],int msgID)
{
        char subject[41],dummy[1];
        parseDate(inBuffer);
        whoFrom(inMsgFp,inBuffer,msgID);

        /*Skip the Blank Line*/
        fgets(inBuffer,MESSAGELENGTH,inMsgFp);

        /*Skip the First TO: line (ignore address to self) */
        fgets(inBuffer,MESSAGELENGTH,inMsgFp);

        getTo(inMsgFp,inBuffer,msgID);
        getCC(inMsgFp,inBuffer,msgID);

        parseAddr(inBuffer,subject,dummy);
        PXCheck(0x330,PXPutAlpha(headerRec,hdrSubjectFld,subject));
        PXCheck(0x331,PXRecAppend(headerTbl,headerRec));

        /* Skip blank lines */
        fgets(inBuffer,MESSAGELENGTH,inMsgFp);
        fgets(inBuffer,MESSAGELENGTH,inMsgFp);
}

void getTo(FILE *inMsgFp,char inBuffer[],int msgID)
{
        char person[MESSAGELENGTH+1];
        char id[MESSAGELENGTH+1];
        while (fgets(inBuffer,MESSAGELENGTH,inMsgFp) &&
                  strstr(inBuffer,"TO") || strstr(inBuffer,"EMS"))
        {
                parseAddr(inBuffer,person,id);
                PXCheck(0x310,PXPutAlpha(routeRec,routeUserFld,person));
                PXCheck(0x311,PXPutAlpha(routeRec,routeIDFld,id));
                PXCheck(0x312,PXPutShort(routeRec,msgMsgNumFld,msgID));
                PXCheck(0x313,PXPutAlpha(routeRec,routeToFld,"TO"));
                PXCheck(0x314,PXRecAppend(routeTbl,routeRec));
        }
}

void getCC(FILE *inMsgFp,char inBuffer[],int msgID)
{
        char person[MESSAGELENGTH+1];
        char id[MESSAGELENGTH+1];
        while (strstr(inBuffer,"CC"))
        {
                parseAddr(inBuffer,person,id);
                PXCheck(0x320,PXPutAlpha(routeRec,routeUserFld,person));
                PXCheck(0x321,PXPutAlpha(routeRec,routeIDFld,id));
                PXCheck(0x322,PXPutShort(routeRec,msgMsgNumFld,msgID));
                PXCheck(0x323,PXPutAlpha(routeRec,routeToFld,"CC"));
                PXCheck(0x324,PXRecAppend(routeTbl,routeRec));
                fgets(inBuffer,MESSAGELENGTH,inMsgFp);
        }
}

void whoFrom(FILE *inMsgFp,char inBuffer[],int msgID)
{
        char person[MESSAGELENGTH+1];
        char id[MESSAGELENGTH+1];

        fgets(inBuffer,MESSAGELENGTH,inMsgFp);
        parseAddr(inBuffer,person,id);
        printf("From: %s\n",person);
        printf("MCI ID: %s\n",id);
        PXCheck(0x350,PXPutAlpha(headerRec,hdrIDFld,id));
        PXCheck(0x351,PXPutAlpha(headerRec,hdrUserFld,person));
        PXCheck(0x352,PXPutShort(headerRec,hdrMsgFld,msgID));
}

void parseAddr(char *inBuffer,char *person,char *id)
{
        char *p;
        char *s = person;
        if ((p = strchr(inBuffer,':'))!=NULL)
        {
                while (*(++p) == ' ' || *p == '*')
                        ;
                while ( (*s++ = *p++) != '/' && *p)
                        ;
                *(--s) = '\0';
        }
        s = id;
        if ((p = strchr(p,':'))!=NULL)
        {
                while ( *(++p) == ' ')
                        ;
                while ( (*s++ = *p++) != '\n')
                        ;
                *(--s) = '\0';
        }
}

short getNextMsgID(short *msgID)
{
        RECORDNUMBER nRecs;
        PXCheck(0x360,PXTblNRecs(msgTbl,&nRecs));
        if (nRecs == 0)
                *msgID = 1;
        else
        {
                PXCheck(0x370,PXRecLast(msgTbl));
                PXCheck(0x371,PXRecGet(msgTbl,msgRec));
                PXCheck(0x372,PXGetShort(msgRec,msgMsgNumFld,msgID));
                ++(*msgID);
        }

        return(*msgID);
}

void parseDate(char *dateStr)
{
        char *mon,*day,*year;
        int iMon,iDay,iYear;
        long engDate;

        strtok(dateStr," ");
        strtok(NULL," ");
        mon=strtok(NULL," ");
        day=strtok(NULL," ");
        year=strtok(NULL," ");
        if((iMon=convertMonth(mon))==0)
                PXCheck(0x380,PXERR_INVDATE);
        iDay=atoi(day);
        iYear=atoi(year);
        PXCheck(381,PXRecBufEmpty(headerRec));
        PXCheck(382,PXDateEncode(iMon,iDay,iYear,&engDate));
        PXCheck(383,PXPutDate(headerRec,hdrDateFld,engDate));
}

int convertMonth(char *str)
{
        switch (str[0])
        {
                case 'A': switch (str[1])
                                {
                                        case 'p': return 4;  /* April */
                                        case 'u': return 8;  /* August */
                                }
                case 'D': return 12;                        /* December */
                case 'F': return 2;                         /* February */
                case 'J': switch (str[1])
                                {
                                        case 'a': return 1; /* January */
                                        case 'u': switch (str[2])
                                                        {
                                        case 'l': return 7; /* July */
                                        case 'n': return 6; /* June */
                                                        }
                                }
                case 'M': switch (str[2])
                                {
                                        case 'r': return 3; /* March */
                                        case 'y': return 5; /* May */
                                }
                case 'N': return 11;                        /* November */
                case 'S': return 9;                         /* September */
                case 'O': return 10;                        /* October */
        }
        return 0;
}
```

End Listing Three

Listing Four

```
#include <stdio.h>
#include <stdlib.h>
#include <pxengine.h>
#include "mci.h"

void createIndex(TABLEHANDLE *tbl,FIELDHANDLE fld[3]);
void processMany(FILE *fp,FIELDHANDLE f[3],TABLEHANDLE t,int pn,char *st);
void processFile(FILE *fp,FIELDHANDLE f[3],TABLEHANDLE t,int pn);
void nextPendNum(TABLEHANDLE t,FIELDHANDLE f,short *pn);

#define PENDFIELD 0
#define ACTFIELD  1
#define TEXTFIELD 2
```

```
void send(void)
{
        TABLEHANDLE tbl;
        FIELDHANDLE flds[3];
        FILE *output;
        int exists;
        short pendNum;
        PXCheck(0x200,PXTblExist("Pending",&exists));
        if(!exists)
                return;
        createIndex(&tbl,flds);
        PXCheck(0x201,PXTblOpen("Pending",&tbl,0,0));
        pendNum=1;
        output=fopen("mci-send.txt","wt");
        if(output==NULL)
        {
                perror("mci-send.txt");
                exit(1);
        }
        do
        {
                fprintf(output,"\ncr\n");
                processMany(output,flds,tbl,pendNum,"TO");
                fputs("\n",output);
                processMany(output,flds,tbl,pendNum,"CC");
                fputs("\n",output);
                processMany(output,flds,tbl,pendNum,"Subject");
                processFile(output,flds,tbl,pendNum);
                fputs("\n/\n\nYes\n",output);
                nextPendNum(tbl,flds[PENDFIELD],&pendNum);
        }while(pendNum>0);

        fputs("exit\n",output);
        fclose(output);
        PXCheck(0x202,PXTblClose(tbl));
        PXCheck(0x203,PXTblDelete("Pending"));
        return;
}
void createIndex(TABLEHANDLE *tbl,FIELDHANDLE fld[3])
{
        PXCheck(0x210,PXTblOpen("Pending",tbl,0,0));
        PXCheck(0x211,PXFldHandle(*tbl,"Pending#",&fld[PENDFIELD]));
        PXCheck(0x212,PXFldHandle(*tbl,"Action",&fld[ACTFIELD]));
        PXCheck(0x213,PXFldHandle(*tbl,"Text",&fld[TEXTFIELD]));
        PXCheck(0x214,PXTblClose(*tbl));
        PXCheck(0x215,PXKeyAdd("Pending",1,&fld[ACTFIELD],SECONDARY));
}
void processMany(FILE *fp,FIELDHANDLE f[3],TABLEHANDLE t,int pn,char *st)
{
        char txtSt[81];
        RECORDHANDLE srchRec,rec;

        PXCheck(0x230,PXRecBufOpen(t,&rec));
        PXCheck(0x231,PXRecBufOpen(t,&srchRec));
        PXCheck(0x232,PXPutShort(srchRec,f[PENDFIELD],pn));
        PXCheck(0x233,PXPutAlpha(srchRec,f[ACTFIELD],st));
        PXCheck(0x234,PXSrchKey(t,srchRec,2,SEARCHFIRST));

        while(PXLastErr()==PXSUCCESS)
        {
                PXCheck(0x235,PXRecGet(t,rec));

        PXCheck(0x236,PXGetAlpha(rec,f[TEXTFIELD],sizeof(txtSt),txtSt));
                fprintf(fp,"%s\n",txtSt);
                PXCheck(0x237,PXSrchKey(t,srchRec,2,SEARCHNEXT));
        }
        PXCheck(0x238,PXRecBufClose(rec));
        PXCheck(0x239,PXRecBufClose(srchRec));
}
void processFile(FILE *fp,FIELDHANDLE f[3],TABLEHANDLE t,int pn)
{
        char txtSt[81],fname[21];
        FILE *inFp;
        RECORDHANDLE srchRec,rec;
        PXCheck(0x240,PXRecBufOpen(t,&rec));
        PXCheck(0x241,PXRecBufOpen(t,&srchRec));
        PXCheck(0x242,PXPutShort(srchRec,f[PENDFIELD],pn));
        PXCheck(0x243,PXPutAlpha(srchRec,f[ACTFIELD],"Filename"));
        PXCheck(0x244,PXSrchKey(t,srchRec,2,SEARCHFIRST));

        while(PXLastErr()==PXSUCCESS)
        {
               PXCheck(0x245,PXRecGet(t,rec));

        PXCheck(0x246,PXGetAlpha(rec,f[TEXTFIELD],sizeof(fname),fname));
                if((inFp=fopen(fname,"rt"))==NULL)
                {
                        perror(fname);
                        exit(1);
                }
                while(fgets(txtSt,80,inFp)!=NULL)
                        fputs(txtSt,fp);
                fclose(inFp);
                unlink(fname);

                PXCheck(0x247,PXSrchKey(t,srchRec,2,SEARCHNEXT));
        }
        PXCheck(0x248,PXRecBufClose(rec));
        PXCheck(0x249,PXRecBufClose(srchRec));
}
void nextPendNum(TABLEHANDLE t,FIELDHANDLE f,short *pn)
{
        RECORDHANDLE r;

        PXCheck(0x250,PXRecBufOpen(t,&r));
        PXCheck(0x251,PXPutShort(r,f,*pn));
        PXCheck(0x252,PXSrchKey(t,r,1,SEARCHFIRST));
        while(PXLastErr()==PXSUCCESS)
                PXCheck(0x253,PXSrchKey(t,r,1,SEARCHNEXT));
        PXCheck(0x254,PXRecNext(t));
        if(PXLastErr()==PXSUCCESS)
        {
                PXCheck(0x255,PXRecGet(t,r));
                PXCheck(0x256,PXGetShort(r,f,pn));
                PXCheck(0x257,PXRecBufClose(r));
        }
        else
                *pn=0;
}
```

End Listing Four

Listing Five

```
editor = "ED"  ; Change this to your word processor

PROC BuildHdrForm()
  {Forms} {Design} {Header} {2} {Brief Header}     ; Design form for HEADER
  " Date Recvd      From                  Subject" ; Place text...
  Enter
  " ----------- -------------------- ----------------------------------------"
  Enter
  " "
  Menu {Field} {Place} {Regular} {Date Received}   ; Place DATE field
  Enter Enter Right
  Menu {Field} {Place} {Regular} {From User}       ; Place FROM USER field
  Enter Enter Right
  Menu {Field} {Place} {Regular} {Subject}         ; Place SUBJECT field
  Enter Enter CtrlHome
  Menu {Multi} {Records} {Define}                  ; Define as MultiRecord
  Enter Left Enter Down Down Down Down Down Down Down Enter
  Do_It!                                           ; Save this form
ENDPROC ; BuildHdrForm

PROC BuildMsgForms()
  {Forms} {Design} {Message} {1} {Message Body} ; Design form for MESSAGE
  MENU {Field} {Place} {Regular} {Line#}        ; Place LINE# field
  ENTER LEFT LEFT LEFT LEFT LEFT LEFT LEFT LEFT LEFT LEFT; Truncate field size
  LEFT LEFT LEFT LEFT LEFT LEFT LEFT LEFT LEFT LEFT LEFT LEFT ENTER

  MENU {Field} {Place} {Regular} {Text}         ; Place TEXT field
  RIGHT ENTER ENTER

  MENU {Multi} {Records} {Define}               ; Define as MultiRecord fields
  ENTER LEFT ENTER DOWN DOWN DOWN DOWN DOWN DOWN DOWN DOWN ENTER

  ; The next four lines set up the position and color for hiding the line#
  ; field. It currently sets the color to a light grey on light grey.
  HOME CTRLHOME
  MENU {Style} {Color} {Area}
  ENTER DOWN DOWN DOWN DOWN DOWN DOWN DOWN ENTER
  RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT ENTER
```

(continued on page 108)

Listing Five *(Listing continued, text begins on page 72.)*

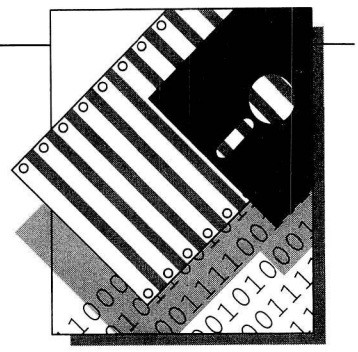
```
  DO_IT!                                          ; Save this form

  {Forms} {Design} {Header} {1} {Message}         ; Design for for HEADER table
  ENTER "  Date Received "                        ; Set up position and text
  MENU {Field} {Place} {Regular} {Date Received}  ; Place DATE RECEIVED field

  ENTER ENTER ENTER  "  Subject ..... "           ; Set up position and text
  MENU {Field} {Place} {Regular} {Subject}        ; Place SUBJECT field

  ENTER ENTER ENTER  "  From ........ "           ; Set up position and text
  MENU {Field} {Place} {Regular} {From User}      ; Place FROM USER field
  ENTER ENTER
  MENU {Multi} {Tables} {Place}        ; Place MultiTable Linked Form from
  {Linked} {Message} {1} {Message#}    ; MESSAGE table.
  UP UP UP UP UP UP UP UP UP ENTER     ; Position the field

  DO_IT!                                          ; Save this form
ENDPROC ; BuildMsgForms

PROC MenuScrn()
  CANVAS OFF
  @4,0
  SETMARGIN 25

  PAINTCANVAS ATTRIBUTE 31 2,0,24,79 ; Fill Background w/White on Blue
  PAINTCANVAS ATTRIBUTE 0 5,27,8,54  ; Shadow Box in Black
  STYLE ATTRIBUTE 7
  TEXT
---------------------------
:    MCI Message System    :
:                          :
---------------------------
  ENDTEXT

  SETMARGIN 0
  @22,0

  TEXT
-------------------------------------------------------------------------------
    Use Cursor Keys <- -> To Move Between Menu Choices, or 1st Character
-------------------------------------------------------------------------------
  ENDTEXT
  PAINTCANVAS ATTRIBUTE 31 22,0,24,79
  CANVAS ON
ENDPROC ; MenuScrn

PROC Inbox()
  IF ( ISEMPTY("Header") ) THEN
    @ 20,27
    ?? "No messages to be viewed"
    RETURN
  ENDIF
  VIEW "Header"
  PICKFORM 2
  WAIT TABLE
    PROMPT "Arrow to message, press RETURN to view.",
           "ESC aborts to main menu."
  UNTIL "Enter", "Esc"

  SWITCH
    CASE retval = "Enter" :
      ViewMessage()
    CASE retval = "Esc" :
      CLEAR
      CLEARALL
      RETURN
  ENDSWITCH
ENDPROC ; Inbox

PROC ViewMessage()
  PICKFORM 1
  DOWNIMAGE
  WHILE (TRUE)
    WAIT TABLE
      PROMPT "F3/F4 move to Previous/Next message. INS responds to message.",
             "Arrow keys scroll the current message. ESC aborts to main menu."
    UNTIL "Esc", "F3", "F4", "Ins"

    SWITCH
      CASE retval = "Ins":
        Respond();
        PICKFORM 1
        DOWNIMAGE
      CASE retval = "F3":
        UPIMAGE
        PGUP
        DOWNIMAGE
      CASE retval = "F4":
        UPIMAGE
        PGDN
        DOWNIMAGE
      CASE retval = "Esc":
        QUITLOOP
    ENDSWITCH
  ENDWHILE
  CLEAR
  CLEARALL
ENDPROC ; ViewMessage

PROC CheckPending()
  IF ( NOT ISTABLE("Pending") ) THEN
    CREATE "Pending"
      "Pending#" : "N*",
      "Action"   : "A8*",
      "Text"     : "A40*"
    RETURN 1
  ELSE
```

(continued on page 110)

Listing Five *(Listing continued, text begins on page 72.)*

```
    IF ( ISEMPTY("Pending") ) THEN
     RETURN 1
    ELSE
     RETURN CMAX("Pending","Pending#") + 1
    ENDIF
  ENDIF
ENDPROC ; CheckPending
PROC Respond()
  UPIMAGE
  mnum    = [Message#]
  subject = [Subject]
  from    = [From ID]
  pnum    = CheckPending()

  QUERY
  Route ¦ Message# ¦ TOorCC ¦ User Name ¦ User ID ¦
        ¦ ~mnum    ¦ Check  ¦           ¦ Check   ¦
        ¦          ¦        ¦           ¦         ¦
        ¦          ¦        ¦           ¦         ¦

  ENDQUERY
  DO_IT!

  VIEW "Answer"
  IF ( ISEMPTY("Answer") ) THEN
    answerTable=TRUE
  ELSE
    answerTable=FALSE
  ENDIF
  VIEW "Pending"
  EDIT "Pending"
  END
  DOWN
  [Pending#] = pnum
  [Action]   = "Filename"
  [Text]     = "Pending." + STRVAL(pnum)
  DOWN
  [Pending#] = pnum
  [Action]   = "Subject"
  [Text]     = "RE:" + subject
  DOWN
  [Pending#] = pnum
  [Action]   = "TO"
  [Text]     = from
  DOWN
  UPIMAGE

  IF ( NOT answerTable ) THEN
    WHILE ( TRUE )
      to_cc = [TOorCC]
      uid   = [User ID]

      DOWNIMAGE
      [Pending#] = pnum
      [Action]   = to_cc
      [Text]     = uid
      DOWN

      UPIMAGE
      IF ( ATLAST() ) THEN
        QUITLOOP
      ENDIF
      DOWN
    ENDWHILE
  ENDIF

  DO_IT!

  RUN BIG editor + " Pending." + STRVAL(pnum)
  CLEAR
  CLEARIMAGE          ; Close the Pending Table
  CLEARIMAGE          ; Close the Answer Table
  UPIMAGE
  CLEARIMAGE          ; Close Route Query Table
ENDPROC ; Respond()

PROC Compose()
  pnum = CheckPending()

  VIEW "Pending"
  EDIT "Pending"
  END
  DOWN

  [Pending#] = pnum
  [Action]   = "Filename"
  [Text]     = "Pending." + STRVAL(pnum)
  DOWN

  CLEAR
  @ 0,0
  GetMultiple("TO",TRUE);
  IF ( NOT retval ) THEN
    CANCELEDIT
    CLEAR
    CLEARALL
    RETURN
  ENDIF
  GetMultiple("CC",FALSE);
  IF ( NOT retval ) THEN
    CANCELEDIT
    CLEAR
    CLEARALL
    RETURN
  ENDIF

  sub = ""
  WHILE( LEN(sub) = 0 )
    ? "Subject: "
    ACCEPT "A40" to sub
    IF ( NOT retval ) THEN
      CANCELEDIT
      CLEAR
      CLEARALL
      RETURN
    ENDIF
  ENDWHILE
  [Pending#] = pnum
  [Action]   = "Subject"
  [Text]     = sub

  DO_IT!

  RUN BIG editor + " Pending." + STRVAL(pnum)
  CLEAR
  CLEARALL
ENDPROC ; Compose

PROC GetMultiple(where,one)
  IF (one) THEN
    name = ""
    WHILE ( LEN(name) = 0 )          ; Loop until at least one [where] field
      ? where + ": "                 ; has been input.
      ACCEPT "A20" TO name
      IF ( NOT retval ) THEN         ; Esc was pressed
        RETURN FALSE
      ENDIF
    ENDWHILE
    [Pending#] = pnum
    [Action]   = where
    [Text]     = name
    DOWN
  ENDIF

  WHILE (TRUE)
    ? where + ": "
    ACCEPT "A20" TO name
    IF ( NOT retval ) THEN           ; Esc was pressed
      RETURN FALSE
    ENDIF
    IF ( LEN(name) = 0 ) THEN
      QUITLOOP
    ENDIF
    [Pending#] = pnum
    [Action]   = where
    [Text]     = name
    DOWN
  ENDWHILE
  RETURN TRUE
ENDPROC ; GetMultiple

PROC Main()
  WHILE (TRUE)
    Menuscrn()

    SHOWMENU
      "Inbox"  : "View Current Messages",
      "Compose": "Compose a new Message",
      "Quit"   : "Quit the MCI Message System"
      TO choice

    SWITCH
      CASE choice = "Inbox" :
        Inbox()
      CASE choice = "Compose" :
        Compose()
      CASE choice = "Quit" :
        SHOWMENU
          "No"  : "Do NOT Quit The Application.",
          "Yes" : "Quit The Application."
          TO theexit
        IF ( theexit = "Yes" ) THEN
          MENU {Exit} {Yes}
        ENDIF
    ENDSWITCH
  ENDWHILE
ENDPROC ; Main

;**** Mainline ****
RESET
CLEAR
CLEARALL

IF ( NOT ISTABLE("Message") ) OR ( NOT ISTABLE("Header") ) OR
   ( NOT ISTABLE("Route") ) THEN
  @ 12,17
  ?? "You must run MCI.EXE before running this script"
  SLEEP 5000
  RETURN
ENDIF

IF ( NOT ISFILE("Header.F1") ) OR ( NOT ISFILE("Message.F1") ) THEN
  BuildMsgForms();
ENDIF

IF ( NOT ISFILE("Header.F2") ) THEN
  BuildHdrForm();
ENDIF

Main()
```

End Listings



Wrapping Up Loose Ends

Michael Swaine

The occasion of the last "Paradigms" column of this year is to tie up some long-dangling loose ends and float some semi-detatched observations on parallel processing, transputers, fractal graphics, multiparadigm programming, and hypertext, and other hyperstuff. And to mention a few books that are unlikely to show up in "Programmer's Bookshelf," but that still may be worth a look, for reasons to be explained herein.

MINOS Addition

Last November, I wrote about the MINOS system, perhaps the earliest practical implementation of neural net technology. I also described an even earlier neural net system in something I referred to as the ADAM system. One reader wrote to ask for sources on the ADAM computer. The body of literature on this device is enormous, but a primary source is Genesis, which, while of disputable reliability and often weak on technical detail, is at least widely available.

Transputer Refuter

Quite some time ago I allowed a probably overly harsh assessment of the transputer architecture to appear in this column. One reader, a former Inmos employee, wrote to take me to task for this, but time and space didn't allow printing his comments then. Belatedly, here is a brief defense of the Inmos transputer:

> The lack of a register add is offset by the RISC-like nature of the stack workspace access being as quick as an R-to-R add in 33nS on-chip memory (the 4 Kbytes are like 1000 word-length registers). The 30MHz T800 processor puts the fastest 68030/68882 pair to shame speedwise, in most applications. . . . I understand that there are plenty of things to be critical of in the transputer, but the structure isn't one of them! I still support the transputer and can honestly say after seeing many benchmarks and comparisons that the transputer definitely keeps up with the other processors and still provides a direct method of parallelization that no other processor can approach.

The latest transputer-based application whose notice crossed my desk is Equator, a 3-D spreadsheet for the Mac II with a transputer board, from Tri-Millennium Corp. (40 Horace Street, Needham, MA 02194). It looks interesting because: Speed and memory size are scalable with additional processors; the spreadsheet will read and save in standard spreadsheet formats; Equator is priced competitively with other spreadsheet software; and it looks like one of the few commercial applications available that could provide some useful real-world benchmarking for parallel processing.

Multiparadigmatic Programming

I have recently had a chance to play around with release 2.0 Prograph from TGSSystem (Halifax, Nova Scotia). Prograph is a Macintosh-based development system that purports to be a multiparadigmatic system, supporting object-oriented programming, dataflow programming, and visual programming paradigms. Well *purports* may be unfair; the product is just described as *an object-oriented visual programming language and software development environment*. Besides, while it does support dataflow specification of program execution, maybe being multiparadigmatic isn't much of a distinction these days. Prograph is interesting, though, from a couple of perspectives.

First, though, the component list and system requirements. Prograph consists of an editor/interpreter for its visual programming language, a compiler, an application builder, and a number of examples and online tutorial aids. It runs on a Mac Plus or above with 2-Mbyte RAM, a hard disk, and System software Version 6 or above.

As soon as you invoke the editor, you're up against the visual programming language, and using it is certainly a different experience from conventional programming. Prograph is not the only visual programming language around, even for the Mac: Prototyper and VIP are competitors. Its approach to visual programming, though, is its own, and my first reaction is positive.

Lots of products and approaches have been labeled as visual programming systems. There are diagrammatic systems that communicate aspects of the structure of the program, or that permit manipulation of that structure, using flowcharts or dataflow diagrams or state transition diagrams; there are table-based and form-based programming systems for producing databases; and there are iconic languages in which icons take the place of linguistic components in conventional languages. Prograph is in the iconic camp, but has characteristics of other approaches. Icons are strung together via dataflow links, for example.

The visual approach in Prograph is not cosmetic: The whole system is fundamentally visual, with multiple windows for editing and executing programs and for specifying classes and methods. Even the debugging environment is visual. The icons and dataflow diagrams constructed from them are not pictures representing blocks of code, but executable items. Click on them and they do their thing. The visual elements really are the program.

So what is it like to program with icons and diagrams? It comes down to a lot of picking items from lists and

pasting them where you want them, at the most mundane level. And typing names into newly created objects. Most of the icons are really just variously shaped text boxes, which is to be expected. They are icons, not Chinese ideographs, and the only way the set of icons can be extensible is through the use of existing linguistic devices, meaning names spelled out with letters. Most of the Mac Toolbox calls are represented iconically and accessed as Mac Methods. Their names are not editable, but are fixed to be the names given for Pascal functions and procedures in *Inside Macintosh*.

Hypertext is creeping into everything I read. There is definitely something there, but the hype can be off-putting

With Prograph I get more of a sense that it works than with other visual programming systems I've looked at. The icons and the dataflow links that tie them together do make a lot of sense. Working with the system actually makes programming feel more like building a structure from components, and, at least in the simple examples I've played with, gives me a clearer sense of the structure of the program than I get from programming in a conventional language. A moderately complex application might be another matter entirely.

I also like the commenting style built into the system. Since the program itself consists of graphical elements — structures of boxes — the snatches of text pasted next to the boxes or in the margins stand out much more than I've seen comments stand out in any conventional code. They are clearly in a different language. The message that comes through is: If it's a block of text, it's a comment.

The dataflow model is inherently concurrent, and it's a fine-grained concurrency based on data availability. That's true of any dataflow system, and it is completely academic on a single processor machine like the Mac, but TGSSystems hints at the likelihood of making use of the concurrency in some future release that would support a parallel processing architecture, though not necessarily in an Apple box. I had wondered how dataflow specification matches with object-oriented programming, and got at least a partial answer in the fact that Prograph allows the data that flows along the links to be instances of object-oriented classes, as well as elementary data types. More than one data item can be passed along a link, with the node receiving this pool of data recognizing the value it needs and selecting it: This is dynamic dataflow.

There's an interesting point about Prograph's positioning. The marketing copy on the box explicitly refers to HyperCard. This is curious, because Prograph is clearly more powerful and more complex than HyperCard and other products of that hyperilk. The point seems to be to project it as falling midway between HyperCard and Pascal or C on some spectrum. TGSSystems calls Prograph a very high-level language and a fifth-generation tool.

Perhaps what they have in mind in invoking HyperCard is the Application Builder. This is a toolkit of editors and OOP classes for building Mac applications. It's purely an alternative for beginners or for those occasions when building from the ground up is not necessary, but it does give Prograph the aspect of an application generator. This should aid the new Mac developer, for whom the advice has always been: First, read *Inside Macintosh Volumes 1-5* (and counting). With Prograph's approach, a developer ought to be able to start programming and refer to IM as needed. And it may explain the reference to HyperCard.

Hypertext and Hyperstuff

One thing that Prograph doesn't claim to be is hypertext. I don't know why not; everybody in the world seems to be jumping on that bandwagon. Hypertext is creeping into everything I read. There is definitely something there, but the hype can be off-putting. The best source I know of on the subject, touching on its roots and underlying philosophy and current implementations, is *Macintosh Hypermedia* by Michael Fraase (Scott, Foresman, 1990). Too bad it's Mac-specific.

I don't recommend the book *The Society of Text* (Edward Barrett, ed., MIT Press, 1989) unless you're really into hypertext. It seems to be oriented toward academics first and people who work with text second; not toward software authors at all. I do recommend one particular article in the book, though, to any readers who suspect that they might find themselves pressed against their will to incorporate a hypertext approach into some software

(continued from page 114)
project. It's "The Missing Link: Why We're All Doing Hypertext Wrong" by Norman Meyerowitz, and it could help to argue your way out of something you don't want to do.

Meyerowitz's view is that hypertext has to be integrated into the operating system before it really makes sense. What we have now, he says, are insular tools that may solve some specific problem but that aren't what he has in mind when he thinks of hypertext. HyperCard may not be the best example of the hypertext state of the art, but it amply demonstrates Meyerowitz's concerns. You can do hypertextish things in HyperCard, but the problem is that you have to be in HyperCard to do them. And what you probably want, or would want if you really knew why you should want to do hypertextish things, is to do them where you are with what you've got.

Meyerowitz, who heads Brown University's Intermedia project in hypertext or hypermedia, has the credentials to make his view of what hypertext is supposed to be worth noting. What he thinks is required is to tie the hypertexting into the operating system, and the way to do that is to add the element of a navigational link at the appropriate operating system level. Meyerowitz takes this to be the level of desktop tools in Windows or the Mac or similar GUI systems. Meyerowitz and his colleagues have actually implemented such links in their Intermedia system, but this is a case of prototyping the future, to use his words. The fact is, existing operating systems don't have such navigational links, therefore they can't support true hypertext, and anything short of true hypertext is just pseudohypertext, or in other words, just the current fad. That, anyway, is an argument that you can use.

When the grumbling stops and everything settles down, we'll have a better idea what HyperCard's future will be

Since I have previously in this column mentioned ToolBook, Asymetrix's HyperCard-like product for Windows 3.0, I'll pass on the news that Asymetrix is shipping its ToolBook Author's Resource Kit (ARK) for $450. The ARK includes a master copy of the runtime version of ToolBook and a license to distribute royalty-free copies of the runtime version and the DLLs included with the full version of ToolBook. It also includes tech support, a utility program for examining and changing properties of books you create with ToolBook, online books of ideas on design, and Script Remover, an application that lets you remove the readable source from your ToolBook application before distributing them. The telephone number, if you want to order the ARK or to find out about the developer program, is 206-637-1500.

And while we're on the subject, HyperCard itself will, I suspect, begin to evolve into clearly distinguished user and developer versions, now that it is a Claris product.

A poorly written press release resulted in a *MacWeek* story to the effect that Apple would be selling a crippled version with new systems, basically a runtime HyperCard, with the developer version available at a *competitive* price. Since anything that could be considered competitive with HyperCard costs

(continued from page 116)
an order of magnitude more than HyperCard, and since the HyperCard user community felt that it had a moral commitment from Apple that HyperCard would continue to be distributed like system software, this did not go over well. I read the press release, and the confusion was all Apple's fault.

The true story is — but come to think of it, I only know what the true story is this week. Apple's reingestion of Claris seems not to be proceeding altogether smoothly, and there may be further burps. When the grumbling stops and everything settles down, we'll have a better idea what HyperCard's future will be. It appears, though, that Claris knows that if it is going to create two versions of HyperCard, it should do so by adding functionality to the developer version, not by removing it from the user version.

Holiday Reading

I have discussed fractal flora here in past columns. Anyone interested in pursuing the topic of how fractal graphics can be applied to the study and simulation of plant structure and growth should get hold of *The Algorithmic Beauty of Plants*, by Przemyslaw Prusinkiewicz and Aristid Lindenmayer (Springer-Verlag, 1990). It's a beautiful book, as well as a good presentation of the major algorithms in this field. It is based on Aristid Lindenmayer's L-systems approach, which I was surprised to learn dates back to 1968. The book deals with modeling plant development, particular plant organs and cellular structures, as well as producing animated displays of plants growing. The dust jacket describes the subject matter as *an attempt to re-create the structures of life*, and that seems pretty accurate to me.

The authors supply enough theory and pseudocode to make their points clear, but there are no type-and-run examples. In fact, the system that the authors used to produce the beautiful plant forms reproduced in the book is not something they'd be likely — or able — to reproduce in the book. It sounds interesting enough to justify a book on its own. The authors refer to the system as a virtual laboratory, *a microworld which can be explored under the guidance of a hypertext system*. Yeah, hypertext again. By microworld, they mean *an interactive environment for creating and conducting simulated experiments*.

They got the basic ideas for their virtual laboratory from Ted Nelson's *Computer Lib/Dream Machines* (their reference: Ted Nelson, 1974; latest edition, Tempus Books, Microsoft Press, 1987). Ted Nelson, as the whole world knows, coined the term hypertext and has devoted more energy to its achievement than anyone else in the world. And, just to make sure I get all the links linked, I should mention that Norman Meyerowitz took over Brown University's hypertext research from Andries van Dam, who codesigned the world's first hypertext editing system in 1967 with Ted Nelson and a group of Brown undergraduates.

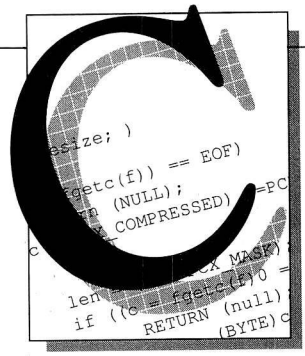
Wherever they got the idea, the virtual laboratory sounds nifty. I wish someone would buy me one for Christmas. The book is pretty enough to be a gift, though. See if you can get someone to buy it for you.

Shared Minds by Michael Shrage (Random House, 1990) is a book worth taking a look at for what it has to say about the future direction of software development. It's a nontechnical treatment of the phenomenon of collaborative computing: groupware and suchlike stuff. What Shrage has to say about collaborative efforts, particularly in science, runs counter to a widespread spirit of do-it-yourself-ism in the personal computer field. As someone who has been in deep retreat from collaborative efforts for the past three years, I found that it made me think a little more deeply about individual vs. group efforts. It may make you question some of your deeply held rugged individualism, or it may make you mad, or it may make a lot of sense. In any case, it makes good leisure-time reading. *The Mac Is Not a Typewriter* by Robin Williams (a different Robin Williams) (Peachpit Press, 1990) is not a book that *DDJ* readers need to read, but it is an excellent gift book for anyone you know who is nontechnical and is getting to know the Macintosh. It's filled with simple but useful advice on producing printed output from the Mac that even professionals may not handle well. Things like not double-spacing after periods, using real quotes and apostrophes, handling punctuation and parentheses, advice on typefaces: It's not a complete treatise on grammar or type handling, but a wise selection of the most helpful tips for the broadest possible audience. It's a quick read, and only costs $9.95. Give it to someone you like about that much.

See you next year!

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 10.**



The B-tree Again

Al Stevens

In the June 1990 issue on hypertext, I published a program that used a subset of the B-tree algorithm to build a HyperTree index into a text file. That algorithm lacked many of the features of a complete B-tree, because it was dedicated to a single purpose, the construction and search of a static index into text. The B-tree code I used was an adaptation of some B-tree algorithms from a book I wrote about using the C language to describe, build, and maintain databases. I am writing a second edition of the book now to upgrade the code and techniques to the facilities of ANSI C, and this column is a preview of the B-tree code as it will appear in the new work.

You don't need all the code in the book to use these algorithms. As with most of the code I develop and publish, the B-tree functions are standalone C tools for you to use anywhere you need a B-tree solution.

What is a B-tree?

The B-tree algorithm was developed in 1970 by R. Beyer and E. McCreight. It is a tree-structured index mechanism that is balanced and that is self-tending. First some definitions. (My pal, Tommy Saunders, the great Dixieland cornet player, tells a story about bee trees. These B-trees are different, Tommy.)

A "tree" is a multilayered hierarchy of "nodes." There is one "root" node at the top. The root is the "parent" node to the "child" nodes at the next level down. Nodes contain key values which deliver corresponding data values. Each parent node can have multiple child nodes, and any child node can have no more than one parent. The nodes at the bottom of the hierarchy contain all the data values for the keys, even if the key values are in nodes at higher levels. Therefore, the data values themselves are the "leaves." Usually the leaves are vectors to records in a file where the data elements associated with the key values exist.

A balanced tree is one where all the paths from the root to the leaves descend the same number of levels. A self-tending tree is one where the tree balance is maintained by the addition and deletion algorithms; the tree needs no extra tree-balancing procedures.

A B-tree node contains a fixed maximum number of keys referred to as *m*. The algorithm assures that each node, except the root, will have at least *m*/2 keys. The root can have from one to *m* keys. When the root is empty, the tree is empty. When the root is the only level of the tree, the root itself contains the leaves.

The B-tree has another interesting property. You can search the tree for an arbitrarily chosen key value. This search is random retrieval. You can navigate the same tree in the sequence of the keys in either direction starting from any given key. This search is sequential retrieval. You can, therefore, locate a chosen key value and proceed from there through the keys in ascending or descending sequence. This mix is often called Indexed Sequential Access Method (ISAM), a phrase I first heard in the 1960s used for mainframe Cobol files. The B-tree algorithms postdate ISAM, but many ISAM implementations use B-trees. Figure 1 illustrates a simple B-tree. It uses letters of the alphabet as keys and the words of the phonetic alphabet as data values. In this example, *m* is 2, but most B-trees will have a much higher *m* value.

A node contains key values and vectors. The vectors in the root and subordinate nodes are pointers to nodes in the next lower level in the tree. The vectors in the lowest level are the leaves, and these usually point to the data records. The keys in each node are in ascending collating sequence so that a search can proceed from left to right. Some B-tree implementations have large *m* values and use a binary search to search each node.

A node has one more vector than it has key values. Each key value has a vector that points to the nodes with key values less than this one and another vector that points to the key value greater than this one. Look at Figure 1. The root node has the key value "I" and two vectors. The first vector points to the node with keys "C" and "F" while the second vector points to the node with "L" and "O."

The tree in Figure 1 has three levels. The vectors from the third level are leaves that point to the data records that contain the data values (phonetic alphabet words) associated with the keys (letters of the alphabet).

There is room for confusion here. Earlier I said that the leaves were the data values. Now I am saying that they are pointers to the data values. The B-tree delivers its answer in the form of the leaf associated with the key value. The leaf can be whatever you want it to be. But to simplify the architecture of the B-tree, a leaf should resemble a vector so that the format of the nodes at the lowest level is the same as the format of nodes at higher levels. Usually the leaf points to a record in another file, and the file can be in random order. Access to the file is a function of retrievals from the index.

The structure we are discussing looks like an inverted tree. The root is at the

top, and the leaves are at the bottom. Because of this metaphor, such indexes are called "inverted" indexes. As you might expect, you can have more than one inverted index into a data file. Any data element or combination of data elements in a file can become the keys to an index. The leaf vector would point to the file record that matches the key.

A B-tree can support duplicate key values. This feature allows one key value to have several leaves and therefore point to more than one file record. A database record usually has a "primary key" value that uniquely identifies the record. The Employee File has one record per Employee Number. The index for the primary key must not allow duplicate key values because only one employee can have a given Employee Number. But an employee record might also have the Department Number, and you might want to retrieve all the employees for a given department. In that case, the Department Number index into the Employee File must support duplicate values because many employees can work in the same department. A data element with an index that supports multiple key values is a "secondary key."

Searching a B-tree

A B-tree search starts with a key argument and delivers either the leaf or the fact that the argument does not exist

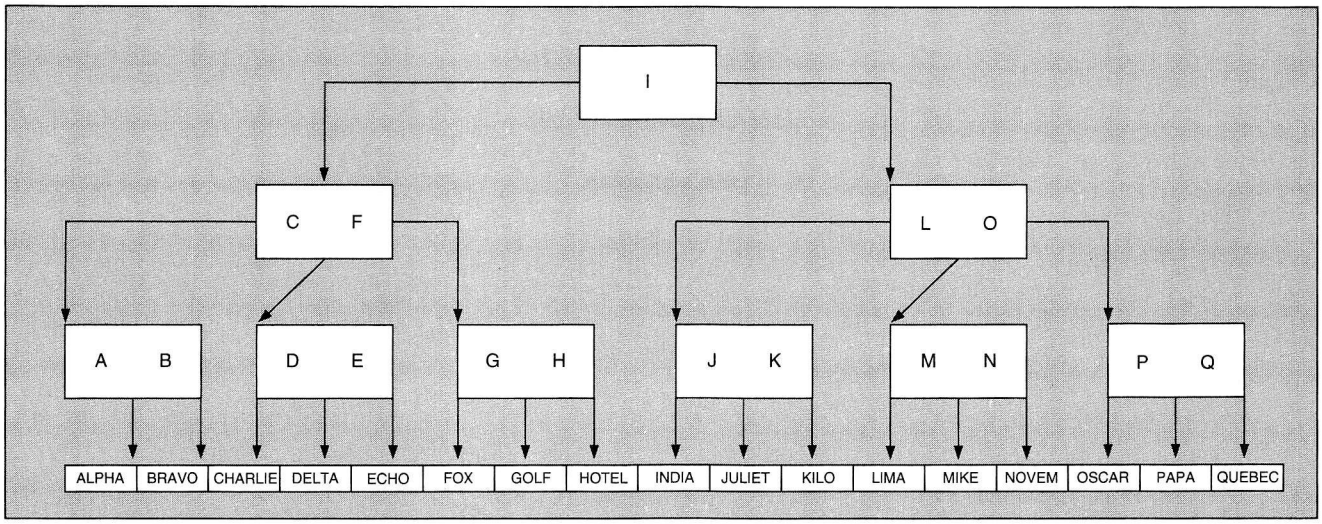


Figure 1: *A simple B-tree*

(continued from page 122)
in the B-tree. Look at Figure 1 again. Every search begins at the root node because the algorithms can always find it. To search for the value *K* you read the root node and search it. If you find the value, you are almost done. If not, you must proceed to the next level. To proceed down, you use the vector that sits after the next lower key and before the next higher key in the current node. In this case, the root has only the value *I*, so, by using the vector just past it, you retrieve the *LO* node in it and begin the search again. This search continues until you either find the value or are at the lowest level. You would find *J* at the bottom level, so the vector just to the right of *J* is its associated leaf and does, in fact, point to the data record that contains the value *JULIET*.

The search is slightly more complex when the argument matches a key in a higher level. If you were searching for *C* the search would stop at the *CF* node on the second level. The vector just past *C* is not a leaf. It points to a lower node. When this happens, you use the vector to read the lower node. The leftmost vector in that node is the one you want. You keep reading nodes by using the leftmost vector until you get to the lowest level. The leftmost vector in the lowest level is the leaf for the argument. Observe that to find *I* you read the *LO* node, use its leftmost vector to read the *JK* node, then use its leftmost vector as the leaf that points to *INDIA*.

A balanced tree is one where all the paths from the root to the leaves descend the same number of levels

Adding a Key to the B-tree

When you want to add a key to a B-tree you must provide the key value and the leaf value to the insertion algorithm. For a database application, you might be adding a record, so the leaf might be the number of the database record.

To add a key you must search for it first. If it exists, you must see if the B-tree supports multiple key values. Some will not, particularly those designated for the primary key of a database file. If the value exists and the index is for unique values, the key addition procedure must return an error indication. If the value does not exist or the tree allows duplicates, you will insert the key in the tree.

Key insertion always begins at the lowest node level. If you found the key in a higher level, you must navigate to the lowest level where the matching key's leaf is kept. If you did not find the key, the search ended in the lowest level at the place where the key goes and that is where you insert the key.

You insert the key and leaf in the node in its collating sequence. If, after the insertion, the node still has *m* or fewer keys, the insertion is done. If, however, the node now has *m*+1 keys, the node must split.

Splitting the node involves building a new node and copying the second half of the keys into it, leaving the first half in the original node. The key at the center of the split must be inserted into the parent of the original node with a vector to the new node. That insertion uses the same code as the first one, so the splitting and growth of the B-tree proceeds upwards to the root. When the root fills and splits, the algorithm builds a new root with only one key.

Deleting a Key from the B-tree

Key deletion always occurs at the lowest level. If the key to be deleted is in a higher node, you navigate down to the lowest node, move its leftmost key value over the top of the key value to be deleted, and then proceed to delete the leftmost key from the lowest-level node.

To maintain tree balance, the delete algorithm looks at the two nodes on either side of the current one at the same level. These nodes are siblings to the current ones. If possible, the algorithm redistributes the keys between the current node and one of its siblings. Further, if the keys of the two nodes would, as a result of the deletion, fit into one node with room for one more, the algorithm combines the keys from both nodes with the key from their mutual parent, and deletes one of the nodes. The key must be deleted from the parent, a procedure that repeats the delete algorithm, allowing the tree to shrink in height as keys depart.

The B-tree Code

There are three source files for the B-tree functions. Listing One, page 142, is database.h, a file that includes those elements from the larger database system that the B-tree functions use. If

Function	Description
*void build_b(char *name, int len)*	This function builds a B-tree file with a file name specified in the name parameter. The *len* parameter specifies the length of a key. The function creates the file, computes and writes all the initial values for the header record, and closes the file.
*int btree_init(char *ndx_name)*	After you have created a B-tree with *build_b*, you can use it in other programs by calling *btree_init*. The *name* parameter matches the name you created the B-tree with. The function returns -1 if the B-tree does not exist, if it is locked, or if you have the maximum number of B-trees already open as defined in btree.h by the global MXTREES variable. Otherwise, the function opens the file, locks it, and returns an integer tree number that you will use in all subsequent function calls related to this B-tree.
int btree_close(int tree)	This function closes and unlocks the B-tree.
*RPTR locate(int tree, char *k)*	This is the key search function. You provide a pointer to a key value, and the function returns the leaf vector if a match exists and a zero vector if it does not. The database.h file defines the RPTR variable type. It is an integral type, and its width defines the range of leaf values.
*int deletekey(int tree, char *x, RPTR ad)*	This is the function to delete a key from the B-tree. You pass a character pointer for the key and the RPTR vector that matches the one you want to delete. This parameter allows the function to differentiate between multiple keys.
*int insertkey(int tree, char *x, RPTR ad, int unique)*	This function adds the specified key value and its RPTR leaf value to the tree. The *unique* parameter is TRUE if the key value must be unique, and FALSE otherwise. If you attempt to add a key that exists when the unique indicator is TRUE, the function returns –1.
RPTR firstkey(int tree) *RPTR lastkey(int tree)* *RPTR nextkey(int tree)* *RPTR prevkey(int tree)*	These four functions navigate the B-tree. The first two functions position the navigation at the beginning or end of the tree in its collating sequence and return the RPTR vectors that match those positions. If the B-tree is empty, these functions return a zero RPTR. The other two functions move either forward or backward to the next sequential key value returning the associated RPTR. You can use these functions following *locate, firstkey, lastkey,* or each other. They return the RPTRs associated with the new positions. If you call *nextkey* when you are already positioned at the last key or *prevkey* when you are already positioned at the first key, the functions will return a zero RPTR.
RPTR currkey(int tree)	This function returns the RPTR for the current key position.
void keyval(int tree, char *ky)	This function copies the key value of the current key position into the string pointed to by the *ky* parameter.

Table 1: *B-tree functions*

(continued from page 124)
we go further into this technology in future columns, we will need to replace this file. Listing Two, page 142, is btree.h, the header file that describes the format of the B-tree records as well as the prototypes for the functions that external programs would call. Listing Three, page 142, is btree.c, the code that implements B-trees.

The delete algorithm looks at the two nodes on either side of the current node at the same level, and redistributes the keys between the current node and one of its siblings

The *treehdr* structure in btree.h describes the header record at the front of every B-tree. The *rootnode* item is a vector to the root node. The *keylength* item is the length of each key. Keys in this B-tree implementation are a fixed length. The *m* variable is the *m* value of the B-tree.

The next two elements, *rlsed_node* and *endnode*, support the file maintenance of the B-tree. Nodes come and go as you insert and delete keys. The *rlsed_node* variable is a vector to the first in a linked list of deleted nodes. When the addition algorithm adds a new node, it takes an entry from this list if one exists. If not, it appends the new node onto the B-tree file and increments the *endnode* variable. This technique reuses deleted nodes to eliminate unnecessary tree growth.

The *locked* variable indicates that the B-tree is in use. The function that initiates B-tree processing sets this flag and writes the header record back to the file. The function that terminates processing clears the flag. Because B-tree integrity is a multirecord matter and because nodes and the header hang around in memory until they need to be written, a system failure could compromise the integrity of the index. The program will not allow you to use a

B-tree that is locked.

The *leftmost* and *rightmost* vectors point to the leftmost and rightmost nodes in the B-tree as viewed in its collating sequence. These vectors allow immediate access to either end of the tree.

The *treenode* structure describes the format of a node record. Each node has some header information of its own followed by the list of keys and vectors. The *nonleaf* variable is zero if the node is at the lowest level of the tree, non-zero otherwise. The *prntnode* vector points to the parent node for this node. If this vector is zero, the node is the root node, the only one without a parent. The *lfsib* and *rtsib* vectors point to the left and right sibling nodes for the current node. The *keyct* variable is the number of keys currently recorded in the node. The *key0* vector is the vector that will be to the left of the leftmost key in the node and that points to the node containing keys less than all keys in this node. The *keyspace* array is the space for the keys and vectors. This space is dynamically distributed according to the key length with room for *m* keys and *m* vectors. The *spil* space is in the structure to allow key insertion to go to *m*+1 keys prior to a split but is not a part of the node in the B-tree file.

The functions in btree.c implement the B-tree algorithms in a way that allows you to have more than one B-tree running at a time. If you use an operating system such as MS-DOS that limits the number of open files for a running program, you might want to modify the code to close and reopen the B-tree files at strategic points.

Table 1 lists the functions that you call from your applications program to use the B-trees. To use the B-tree functions, you must write a program that links with and calls these functions. You must also provide a simple error-processing function named *dberror* that does not return to the caller. There are two error conditions that will invoke *dberror*. One is when the program cannot allocate enough heap. The other is any file error reading or writing the B-tree file. These error conditions place *D_OM* and *D_IOERR*, which are defined in database.h, into the global *errno* variable.

There are many other functions in btree.c, ones that your program does not call but which are used internally by the algorithms. The *btreescan* function scans the tree for a match on the specified key. It modifies pointers in the calling function to indicate where the scan stops and returns TRUE or FALSE depending on whether it found a match or not. It calls the *nodescan*

function, which searches individual nodes. That function calls *compare_keys* to test a key in a node for a match to an argument key. The *fileaddr* function returns the leaf value for the current key position. It calls the *leaflevel* function to navigate down to the lowest level in the B-tree. The *implode* function combines the keys of two sibling nodes, and the *redist* function redistributes the keys of two sibling nodes so that the keys are evenly distributed between the two. The *adopt* function is used by *implode*, *redist*, and *insert_key* to assign a new parent to a child node. The *nextnode* function gets a new node vector from either the released node linked list or from the end of the file to be assigned to a new node. The *scannext* and *scanprev* functions do the tree navigation for *nextkey* and *prevkey*. The *childptr* function reads the parent node of the current one and positions a pointer to the key in the parent node that points to this child. The *read_node* and *write_node* functions do all the node input/output for the functions. They both use the *bseek* function to seek to the node position.

Dear Customer # 00183882

I got my second personal letter from Philippe Kahn today. I could tell it was personal because it was signed in blue ink in Philippe's unmistakable scrawl. The only other celebrity who ever sent me such a personal letter was Jerry Falwell. He wanted money too. I know Philippe's letter was the second one he sent because the manilla window envelope has a big red SECOND NOTICE stamp on it. Philippe thinks that if he makes my neighbors, the mailman, and my bride think I'm a deadbeat who doesn't pay his bills, I'll do whatever he asks to prevent him from sending more letters.

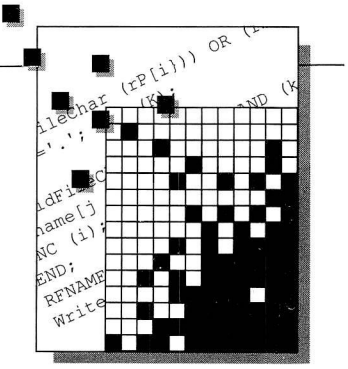
Philippe invited me as a registered user of Turbo C 2.0 to upgrade to Turbo C++ for the pittance of $79.95, a price reduction from his earlier personal letter. He says the price reduction is the result of the overwhelming response from programmers to the first offering. The law of supply and demand has been overturned, I guess. Probably another innovative rendering of the Supreme Court.

I think I'll wait for the THIRD AND FINAL NOTICE so I can get the Ginsu knives.

DDJ

(Listings begin on page 142.)

Vote for your favorite feature/article.
Circle Reader Service **No. 11.**



The Cuba Lake Effect

Jeff Duntemann, KI6RA/7

It was a wonderful trade: Nancy Kress taught me how to write fiction, and I taught her how to use computers. She has gone on to win the coveted Nebula Award for science fiction, and I, well, I've gone on using computers. But now I can write a character that isn't part of the ASCII table. A good deal all the way around.

Nan published a story some years back that still haunts me. In "Down Behind Cuba Lake," the protagonist is travelling through rural New York State to make amends with an old lover with whom she has had — and broken off — a disastrous affair. As night falls she is forced off the main highway by road construction and decides to cut through the back roads near a small lake.

The first back road she takes dead-ends on the shore of the lake. No problem — back roads can be that way. So she backtracks and tries another dark, twisting road — and again, winds up headlights to the water. Again, she backtracks — and again, she winds up somewhere down behind Cuba Lake, on a dead-end dirt road that vanishes into the black water. No matter where she goes, or how far she follows the increasingly narrow and rough dirt roads, she can't get around Cuba Lake. Worse, she can't go back.

It's an exquisite piece of low-key psychological horror. What it means, of course, is that there are some decisions you simply can't rescind, and the more you try, the deeper in trouble you get.

You Can't Get Out

Nan would protest that she doesn't know anything about programming, but anybody who's ever gotten lost somewhere seven levels down in the menu tree of an elaborate transaction-processing application would swear she's been there too.

What I call the "Cuba Lake Effect" is something that almost invariably happens in traditional menu-tree applications that mix modes and menus with data dependencies. You know, things such as this: You're editing a record, and you try to enter a Canadian customer from the Yukon Territory, province code "YT." Whoops, undefined code error — you forgot to add "YT" to the table when you set it up. (Nobody really lives up there, do they? Yup, they do . . . .)

The table maintenance screen is up three levels, across, and down two. You hammer the ESC key, scoot up three screens, and back down into table maintenance . . . but you get stopped cold: You can't edit tables while you're editing records. So back you go, up a screen, across, and down three. Then your blood runs cold as you remember: You can't get out of record edit mode without closing the customer file. You can't close the customer file while the current record edit screen is in an error mode, and you can't close the record edit screen without deleting the current record, and you can't delete the current record without going through the state/province code key file to bring it up on the screen again, and the key for the current record is in an error state with an undefined value . . . .

Get me out of here!

Who Chooses the Path?

This sort of nightmare is one major reason that office staffers traditionally hate computers. What I've described is a common situation on traditional mainframe and minicomputer applications. Such applications begin with a top-level menu screen. The user picks a number, and goes "down" into the next level, typically another menu, and continues until some sort of data entry screen or report definition screen is found. There are a number of paths from the top level down to the leaves of the menu tree, but every path is defined by the structure of the menus, and may be constrained by modes and data dependencies. If you change something the return path may change. It may even be blocked.

The essence of a system such as this (which is about as hideous a way of programming as I've ever seen) is that every path is dictated to the user. Navigating through the program is done strictly on the program's terms. In a badly-designed program (which includes most minicomputer and nearly all mainframe software I've experienced) the program's terms don't necessarily include the imperative that the user be able to back gracefully out of any operation without having to yell for help or abort the whole session.

The essence of writing ergonomic software is giving the user more control of what's going on. This begins with providing the user near-absolute control over how to navigate the system. Ideally, this should be the difference between having to follow the roads on a map to get from point A to point B, or simply having to put your finger over point B on the map and saying, "I want to be here." It is the difference between the view from the inside and the view from a height; in the country of the flat, the 3-D view is king.

Event-Driven Architectures

If you've lived your programming life creating endless full-screen menu trees, your first impression might be that this is impossible. It's not impossible at all,

but like object-oriented programming, such an ideal requires a whole different way of thinking about the architecture of an application. This new architecture was invented by Xerox, borrowed and popularized by the Macintosh (to the extent that Apple has now convinced itself that Apple did the inventing) and is now (with Microsoft Windows at the vanguard) percolating through most commercial applications for DOS. It's called an *event-driven architecture*, and if you're not using it yet you're already way behind the times.

Rather than being a maze of full-screen menus, an event-driven application is typically a single screen with two major parts. At the center of the application (and usually at the center of the screen) is the workspace. This contains some representation of the work that is being done. It might be a technical drawing, a spreadsheet, a document, or a database arranged as rows and columns of records and fields. All around the workspace are tools that may be selected to manipulate the workspace and the information within it.

It always pays to have a handle on what's going on beneath the surface

The user selects tools either from the keyboard, with some combination of control keys, or (much more appropriately) with a mouse cursor that zips around the screen and allows the user to "click on" a tool when the mouse cursor is over that tool's representation on the screen. This representation might in fact be a graphics icon, but remember that nothing in an event-driven architecture limits us to a graphics environment. The new Turbo C++ and Turbo Pascal 6.0 environments are mouse-based and truly event-driven, yet both run strictly in text mode.

A Natural for Windows

Nor is there anything in an event-driven architecture that requires the use of windows, but windows are a natural way to return a measure of control to the user in an event-driven system. Think back to the transaction processing nightmare I described a little earlier. In an event-driven system, the user would be editing records in the central workspace of the application. (Editing records is, at the heart of it, what such transaction processing applications are about.) If an invalid state code turned up, the user would simply reach out with the mouse to a tools icon or menu of some sort, "grab" the codes' editor, and open it as a window, either beside or even on top of the record being edited in the central workspace.

You need to think of the user's action as the equivalent of a subroutine call in the work flow. While editing a record, programmers find the need to do some code table editing. Rather than close the edit screen and backtrack through the menu tree to open another screen devoted to code table editing, the user simply "ducks out" from record editing for a while and opens a window into the code tables. Once done with the code tables, you close the code table tool window and you are instantly back where you were in the central workspace when you discovered the undefined state code. No backing up, no opportunity to get lost behind Cuba Lake. When you find a need for a tool, you reach out and grab it. While you're using the tool, your previous work remains, perhaps dormant, in the central workspace. That's the metaphor of an event-driven architecture.

Beneath the Surface

I'm not going to tell you how to code up an event-driven application framework here. It's probably too big a project for a magazine article, and to be honest, 99.999 percent of you would be wasting your time trying to roll your own, because there are far, far better ones on the open market right now than you could ever hope to create yourselves. They're inexpensive, too — in fact, one of the very best now comes bundled free with every copy of Turbo Pascal 6.0.

Still, it always pays to have a handle on what's going on beneath the surface. So I'll speak in broad terms of how event managers work.

First, to define this thing called an "event:" An event is an occurrence that effects your program but which is not dictated by the flow of control in the program. In other words, an event is not something triggered by an IF statement. It is triggered, instead, by forces outside of the current program, and even outside of the machine itself.

The best example of an event is something that the user does. A keystroke is perhaps the crispest example of an event. The program has no idea what the user will type, or when. Similarly, the user's moving a mouse cursor over a predefined "hot area" (such as a menu bar) is something done when the user chooses to do it. A mouse button click, like a keystroke, is another excellent example of an event.

Events can also come from within the machine. A character from a serial port can be an event, as one can appear at any time once the port is enabled. The system software (DOS, BIOS) can also create events. The best known of these is the "timer tick" that happens 18 times every second. Timer ticks are not beholden to the currently-running program. They march inexorably in step with the hardware timer regardless of what the current program does (unless, of course, the current program inhibits interrupts). Usually, timer tick events don't happen every tick, but every *N*ticks, providing a programmable-

(continued from page 132)
time delay between events.

Similarly, timer ticks can be used to create "now is the time" events; you set a specific date and time when something must happen, and at each timer tick, a simple comparison determines whether it's zero hour or not. If the time is now, the event is generated.

Event Managers

Beneath the surface of an event-driven application must be some sort of event manager. The event manager is a mechanism that watches for events, (typically by "capturing" an interrupt vector signalling an event's occurrence) places them in an application-friendly "wrapper" of some sort, and then queues the wrapped events up for processing in some sort of FIFO (First In, First Out) data structure. The event manager then processes the queued events in order.

Nearly all event-driven systems have this much in common. How the queued-up events are processed is up to the cleverness of the system architect, and varies with the event-driven system or toolkit under discussion. Typically, the programmer associates an event with a procedure, function, or object method that is called when the event occurs. Turbo Pascal 4.x and later, for example, considers function key F10 an event. When the F10 event is detected, Turbo Pascal calls its menuing system and moves the cursor to the menu bar at the top of the screen.

Events can be prioritized, meaning that some events are handled immediately even if others are waiting in the queue. The word "queue" suggest a whole conga-line of events in a row struggling toward their ultimate fate, but that's misleading. In nearly all cases, the event queue is empty or contains one or (on the outside) two events. A good system should not allow events to pile up; there's nothing like watching a system try to execute a queue of a dozen accumulated mouse clicks to drive a user to distraction. If something is begun that absolutely cannot be interrupted, (like saving the work file to disk) event sensing should be suspended until — but only until — the critical task is finished.

More typically, an event "freezes" the current task and opens something new in a process akin to (as just mentioned) a subroutine call. You can duck out of editing the work file in Turbo Pascal by pressing any of numerous hot keys such as F10, Alt-F, Alt-C, and so on, each of which generates an event. The edit screen waits patiently while you fool around in the menus; when the menu work is done, the focus returns to the edit file exactly where you left off when you triggered one of the keyboard events.

Event-driven architectures facilitate a level of concurrency impossible in menu-tree systems. Even if the underlying platform isn't multitasking in nature, having three windows open on the screen with an edit file in each is nearly as useful, especially if you can zip from one to another with a single click of the mouse on the selected window.

We're coming to the close of another era in structured programming: The era of doing it all yourself

Getting Involved

People who have been using event-driven systems for years will find all this stultifyingly obvious, but a great many people (particularly programming newcomers) haven't caught on. I think a good many of those holding back assume you have to come up with the event management and menuing code yourself, which simply isn't true. There are a dozen or more solid event-driven application framework products out there, with more appearing all the time.

Most of them, as you might imagine, are limited to C, and many operate only in graphics mode. Still, some of the best support text mode and are in fact available for Pascal. (The best one I've seen for any language, The Zinc Interface Library, is an SAA-compliant C++ product that operates interchangeably in either graphics or text mode; if you're working in C++, I powerfully suggest that you look into it.) TurboPower Software's Object Professional is one that I've mentioned in the past, and will mention again.

Turbo Vision

But the one that most people will be talking about for a while is the event-driven application framework provided as part of the recently released Turbo Pascal 6.0. Borland's Turbo Vision is an "empty application" in object form that you inherit and extend using object-oriented techniques.

Turbo Vision provides a classic, mouse-based windowing user interface

in text mode. In fact, the Turbo Pascal 6.0 integrated development environment (IDE) itself was written using Turbo Vision, so if you want to see what a Turbo Vision application looks like, just look at Turbo Pascal.

Turbo Vision's *TApplication* object type and the object types it includes already contain thousands of lines of code implementing event management, menuing, and windowing, as well as dialog boxes, buttons, pick lists, and other controls. Your part in the programming process is to define a child class of *TApplication* and add the code specific to your own application.

If your application is fairly simple, you don't even have to write the main program. These three method calls comprise the main program of most Turbo Vision applications, and all three are inherited whole from *TApplication*:

```
BEGIN
   MyApp.Init;
   MyApp.Run;
   MyApp.Done
END.
```

Init is the application's constructor, and *Done* is its destructor. You set the application up, you run it, and you shut it down. This is structured programming with a vengeance.

Like its elder cousin Object Professional, Turbo Vision defies easy description in a 3500-word magazine column. The best I can do this time is give you a flavor for how it works, and show you a simple program that uses it. In future columns I'll be speaking of Turbo Vision regularly, now that it's an intrinsic part of Turbo Pascal.

An Experimental Desktop Manager

In only an hour or two, I was able to come up with the prototype of a desktop manager in Turbo Vision, and I've presented it in Listing One (page 151). It's a prototype because what's there is a menu structure and a couple of dialog boxes; there's no real "guts" to the program. That's by design, because Turbo Vision is a user interface tool, and I wanted to show how it helps you put together an event-driven user interface. The guts are a separate issue.

Working with Turbo Vision begins with the definition of your application object, which is always a child class of the abstract class *TApplication*. I called mine *TDesk* (the "T" prefix means "type," and is an Object Pascal convention going back to Apple's Lisa days). You must override three *TApplication* methods: *InitMenuBar*, which sets up menus in the menu bar at the top of the screen; *InitStatusLine*, which sets up any prompts in the status line at the bottom of the screen; and *HandleEvent*, which is your application's event handler, and in many ways the key to the whole process.

Your own code never actually calls any of these three methods. The application constructor *Init* calls *InitStatusLine* and *InitMenuBarthem* while it's setting up the program. *HandleEvent* is called from within *TApplication's Run* method, which is inherited unchanged by your own specific application class.

This struck me very oddly at first, but like OOP in general, it gets more natural the more you work with it. Working with Turbo Vision might be considered a process of setting up a series of effects, and allowing Turbo Vision's internals to provide the causes.

When you call the *Run* method, what happens is that execution vanishes into Turbo Vision (for which source code is not available, sadly) and waits for events to happen. When an event happens, Turbo Vision gets first crack at handling the event. (Some events, like the Alt-X hotkey for exiting an application, are pre-coded into Turbo Vision and inherited by your own application class.) If the event is not something Turbo Vision handles, it passes the event on to your application by calling your application's *HandleEvent* method. In-

side your *HandleEvent* method, you parse the event passed along by Turbo Vision, typically through a CASE statement. If the event provided by Turbo Vision is something your application knows how to handle, the CASE statement calls an appropriate method or other routine to service the event. If the event is undefined, you have the option of safely ignoring it. See Figure 1.

This, at the highest level, is how the flow of control goes in a Turbo Vision application: Events happen. Turbo Vision queues them up, and gets first shot at processing them. If it cannot process an event, Turbo Vision passes the event along to your application. Once your application processes the event, control eventually falls back into Turbo Vision to wait for the next event.

For simplicity's sake I have created only one method that handles an event in JXDESK: *SystemStatistics*, which is called in response to the Statistics option in the System menu. The call is made from the *HandleEvent* method. Note that the event is passed out from within Turbo Vision as a named constant, *SysStatCmd*, which is simply a name given to the value 102. Such a constant is called a "command" in Turbo Vision. You define commands and associate them with events by passing a command to Turbo Vision along with a menu option definition, when *InitMenuBar* is called. Note that all the *NewItem* calls made from within *InitMenuBar* associate their menu items with the null command except for one:

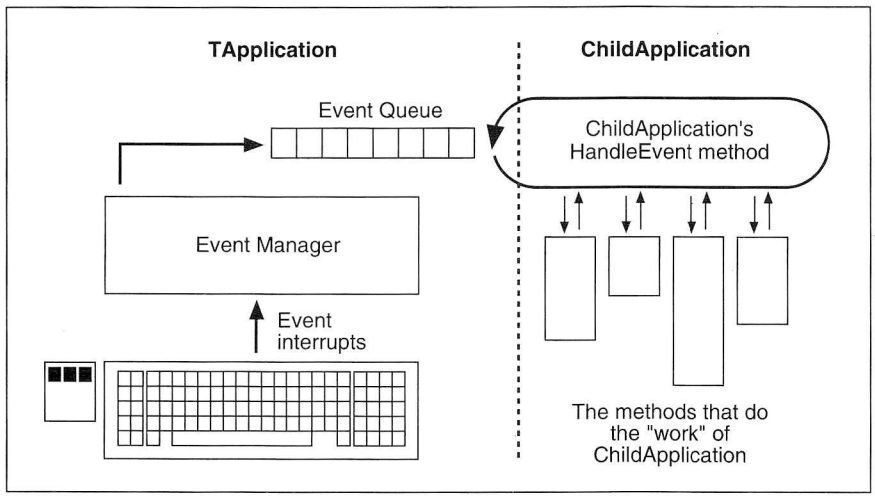


Figure 1: *Turbo Vision event processing*

Products Mentioned

Turbo Pascal 6.0
Borland International
1800 Green Hills Road
Scotts Valley, CA 95066
408-438-8400
$149.95
Turbo Professional $299.95

The Zinc Interface Library
Zinc Software Incorporated
405 South 100 East
Pleasant Grove, UT 84062
801-785-8900
$199.95

The one that sets up the Statistics menu item on the System menu.

Read the code for JXDESK a few times, and it should start making sense. Better still, download the .EXE file that I have provided along with the source to JXDESK and play around with it for a while. Like many other things in our complicated world, Turbo Vision shows better than it tells.

The End of an Era

I'll have more to say about JXDESK in my next column, and more about Turbo Vision as well. And I'll wrap this month up by positing that we're coming to the close of another era in structured programming: The era of Doing It All Yourself. Everywhere I look, Pascal and Modula-2 programmers are using screen generators, procedure and object libraries, menu generators, and all other manner of productivity enhancers and canned code. Nobody but the rankest beginners or the hardest-headed compulsives sit down with the naked compiler and roll it all from scratch.

This makes the apps happen a lot faster, but there's a whole crop of new dangers heaving over the horizon, stemming from not really knowing how your "own" code works. Turbo Vision is a case in point; from now on, as much as half of your application will probably remain a total mystery from a source code standpoint, rather than just some canned procs here or there. You can even get away without learning in detail what Turbo Vision does, as long as you can set up menus and an event handler.

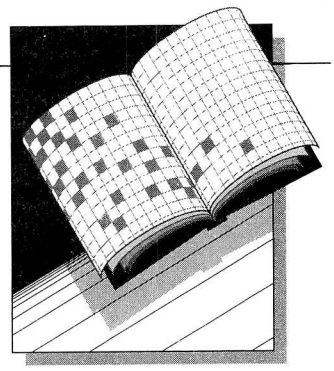
Is this good? I don't know yet. I haven't the vaguest idea how DOS works, and don't care much, because it hasn't hurt me. This wasn't always the case; back in the CP/M era we had to understand what the OS did, because we had to modify and extend it to mate with our oddball hardware. And before CP/M, it was every hacker for himself. I even wrote my own operating system once, lordy.

Now, more and more of the code we use in creating our applications comes standard in a can from somebody else and can be taken for granted once we learn how to use it. Obviously, we're heading somewhere. When I figure out where, I'll be sure to let you know.

DDJ

(Listing begins on page 151.)

Vote for your favorite feature/article.
Circle Reader Service **No. 12.**



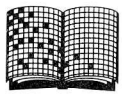
Taking Care of Business

Ray Duncan

Have you ever noticed how, on certain workdays, the wheels of progress seem to turn backwards? Well, I've had entire months like that. If *Peopleware*, by Tom DeMarco and Timothy Lister, had been published about ten years earlier, I might have been saved from a lot of aggravation. (Assuming, of course, that I had the good fortune to stumble across the book, and then had the good sense to take it to heart.)

During my first few years as a full-time software developer and vendor, I made a concerted effort to be instantly available to my customers by phone. I tried to keep tabs on every facet of my company's operation, from managing inventory, to ad design and placement, to paying the bills. I selected an office suite with a few large rooms rather than many small rooms, free passage between the rooms, and little in the way of outside views, under the assumption that this would improve communication and minimize distractions. Ah, the follies of youth!

As the years went by, working conditions at my little company evolved. I moved the firm to another building, where each employee could have a sunny, private, and above all, quiet office. I arranged to screen out nearly all incoming phone calls, diverting technical support questions to electronic mail, a 24-hour electronic bulletin board, or a FAX machine. And I learned to delegate routine decisions about product promotion, purchasing supplies, and triaging of invoices to a trustworthy office manager.



Peopleware: Productive Projects and Teams

Tom DeMarco and Timothy Lister
New York, N.Y.: Dorset House Publishing Company, 1987
188 pages, softcover, $25.00
ISBN 0-932633-05-6

What happened? Over a long period of time, I independently derived one of *Peopleware*'s many axioms: "There are a million ways to lose a work day, but not even a single way to get one back." I also became alerted to the destructive influence of ambient noise and particularly of that demonic invention, the telephone. When I began to keep a call log, I realized how a relatively few telephone conversations, scattered through the course of a normal business day, could reduce the number of lines of new code generated that day nearly to zero. *Peopleware* describes and explains this phenomenon all too clearly:

> During single-minded work-time, people are ideally in a state that psychologists call "flow." Flow is a condition of deep, nearly meditative involvement. In this state, there is a gentle sense of euphoria, and one is largely unaware of the passage of time: "I began to work. I looked up, and three hours had passed." There is no consciousness of effort; the work just seems to, well, flow. You've been in this state often, so we don't have to describe it to you.

Not all work roles require that you attain a state of flow in order to be productive, but for anyone involved in engineering, design, development, writing, or like tasks, flow is a must. These are high-momentum tasks. It's only when you're in flow that the work goes well. Unfortunately, you can't turn on flow like a switch. It takes a slow descent into the subject, requiring fifteen minutes or more of concentration before the state is locked in. During this immersion period, you are particularly susceptible to noise and interruption. A disruptive environment can make it difficult or impossible to attain flow.

Once locked in, the state can be broken by an interruption that is focused on you (your phone, for instance) or by insistent noise ("Attention! Paging Paul Portulaca. Will Paul Portulaca please call extension . . . "). Each time you're interrupted, you require an additional immersion period to get back into flow. During this immersion period, you're not really doing work.

If the average incoming phone call takes five minutes and your reimmersion period is fifteen minutes, the total cost of that call in flow time (work time) lost is twenty minutes. A dozen phone calls use up half a day. A dozen other interruptions and the rest of the work day is gone. This is what guarantees, "You never get anything done around here between 9 and 5."

(continued from page 139)

DeMarco and Lister are justly famous for their texts and seminars on project management, structured programming, and software metrics. But in this slender, witty, and engaging book, they have put aside cold statistics and dataflow diagrams to concentrate on the human aspects of software development: Why some people are productive and others aren't; why some teams jell and others don't; why some strategies to increase quality and beat deadlines have the opposite effect; and why some working environments are conducive to timely, error-free programming and others hinder it. In many ways, *Peopleware* is a New Age counterpart of Frederick Brooks' legendary book, *The Mythical Man-Month.*

Most of the guidelines in *Peopleware* arise from the authors' personal experience, managing projects or consulting on project management for America's largest corporations. Other recommendations are derived from analysis of the "Coding War Games," an annual public competition which pits teams of implementors from different organizations against each other in the design, coding, and testing of a medium-sized program to a fixed specification. Overall, the book is written from an administrator's viewpoint, and directed at an administrator's concern. But there's hardly a page that wouldn't interest the individual programmer and consultant just as well. The book won't change your life, but it may improve it.

If I have a single gripe with *Peopleware*, it's that the book seems to be designed for the MTV generation. It's elaborately ornamented with amusing anecdotes, cute subtitles, aphorisms, and quotations, and it takes pains never to exceed what it assumes to be its reader's limited attention span. In truth, *Peopleware* often seems less like a book than a collection of 25 essays-as-soundbites, each treating a discrete topic ("The Hornblower Factor," "Open Kimono," "Brain Time versus Body Time," "The Telephone," and so on) in five or six pages. I repeatedly found myself wishing that the authors would flesh out a particular concept or case study in, say, an additional 10,000 or 20,000 well-chosen words! Of course, this is less an indictment of *Peopleware* than a sad commentary on the book's intended audience.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 13.**

Listing One *(Text begins on page 121.)*

```
/* --------------- database.h ---------------------- */
#define MXKEYLEN 80  /* maximum key length for indexes     */

#define ERROR -1
#define OK 0

#ifndef TRUE
#define TRUE 1
#define FALSE 0
#endif

typedef long RPTR;  /* B-tree node and file address     */
void dberror(void);

/* ---------- dbms error codes for errno return ------ */
enum dberrors {
   D_OM,      /* out of memory                          */
   D_IOERR    /* i/o error                              */
};
```

End Listing One

Listing Two

```
/* --------------- btree.h ---------------------- */
#define MXTREES 20
#define NODE 512       /* length of a B-tree node          */
#define ADR sizeof(RPTR)

/* ---------- btree prototypes -------------- */
int btree_init(char *ndx_name);
int btree_close(int tree);
void build_b(char *name, int len);
RPTR locate(int tree, char *k);
int deletekey(int tree, char *x, RPTR ad);
int insertkey(int tree, char *x, RPTR ad, int unique);
RPTR nextkey(int tree);
RPTR prevkey(int tree);
RPTR firstkey(int tree);
RPTR lastkey(int tree);
void keyval(int tree, char *ky);
RPTR currkey(int tree);

/* --------- the btree node structure -------------- */
typedef struct treenode    {
   int nonleaf;      /* 0 if leaf, 1 if non-leaf          */
   RPTR prntnode;    /* parent node                       */
   RPTR lfsib;       /* left sibling node                 */
   RPTR rtsib;       /* right sibling node                */
   int keyct;        /* number of keys                    */
   RPTR key0;        /* node # of keys < 1st key this node */
   char keyspace [NODE - ((sizeof(int) * 2) + (ADR * 4))];
   char spil [MXKEYLEN];  /* for insertion excess */
} BTREE;

/* ---- the structure of the btree header node --------- */
typedef struct treehdr    {
   RPTR rootnode;           /* root node number     */
   int keylength;           /* length of a key      */
   int m;                   /* max keys/node        */
   RPTR rlsed_node;         /* next released node   */
   RPTR endnode;            /* next unassigned node */
   int locked;              /* if btree is locked   */
   RPTR leftmost;           /* left-most node       */
   RPTR rightmost;          /* right-most node      */
} HEADER;
```

End Listing Two

Listing Three

```
/* ---------------------- btree.c ----------------------- */
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include "database.h"
#include "btree.h"

#define KLEN bheader[trx].keylength
#define ENTLN (KLEN+ADR)

HEADER bheader[MXTREES];
BTREE trnode;

static FILE *fp[MXTREES];      /* file pointers to indexes   */
static RPTR currnode[MXTREES]; /* node number of current key */
static int currkno[MXTREES];   /* key number of current key  */
static int trx;                /* current tree               */

/* --------- local prototypes ---------- */
static int btreescan(RPTR *t, char *k, char **a);
static int nodescan(char *keyvalue, char **nodeadr);
static int compare_keys(char *a, char *b);
static RPTR fileaddr(RPTR t, char *a);
static RPTR leaflevel(RPTR *t, char **a, int *p);
static void implode(BTREE *left, BTREE *right);
static void redist(BTREE *left, BTREE *right);
static void adopt(void *ad, int kct, RPTR newp);
static RPTR nextnode(void);
static RPTR scannext(RPTR *p, char **a);
static RPTR scanprev(RPTR *p, char **a);
static char *childptr(RPTR left, RPTR parent, BTREE *btp);
static void read_node(RPTR nd, void *bf);
static void write_node(RPTR nd, void *bf);
static void bseek(RPTR nd);
static void memerr(void);

/* -------- initiate b-tree processing ---------- */
int btree_init(char *ndx_name)
{
   for (trx = 0; trx < MXTREES; trx++)
       if (fp[trx] == NULL)
           break;
   if (trx == MXTREES)
       return ERROR;
   if ((fp[trx] = fopen(ndx_name, "rb+")) == NULL)
       return ERROR;
   fread(&bheader[trx], sizeof(HEADER), 1, fp[trx]);
   /* --- if this btree is locked, something is amiss --- */
   if (bheader[trx].locked)    {
       fclose(fp[trx]);
       fp[trx] = NULL;
       return ERROR;
   }
   /* ------- lock the btree --------- */
   bheader[trx].locked = TRUE;
   fseek(fp[trx], 0L, SEEK_SET);
   fwrite(&bheader[trx], sizeof(HEADER), 1, fp[trx]);
   currnode[trx] = 0;
   currkno[trx] = 0;
   return trx;
}

/* ----------- terminate b tree processing ------------- */
int btree_close(int tree)
{
   if (tree >= MXTREES || fp[tree] == 0)
       return ERROR;
   bheader[tree].locked = FALSE;
   fseek(fp[tree], 0L, SEEK_SET);
   fwrite(&bheader[tree], sizeof(HEADER), 1, fp[tree]);
   fclose(fp[tree]);
   fp[tree] = NULL;
   return OK;
}

/* --------Build a new b-tree ------------------ */
void build_b(char *name, int len)
{
   HEADER *bhdp;
   FILE *fp;

   if ((bhdp = malloc(NODE)) == NULL)
       memerr();
   memset(bhdp, '\0', NODE);
   bhdp->keylength = len;
   bhdp->m = ((NODE-((sizeof(int)*2)+(ADR*4)))/(len+ADR));
   bhdp->endnode = 1;
   remove(name);
   fp = fopen(name, "wb");
   fwrite(bhdp, NODE, 1, fp);
   fclose(fp);
   free(bhdp);
}

/* --------------- Locate key in the b-tree -------------- */
RPTR locate(int tree, char *k)
{
   int i, fnd = FALSE;
   RPTR t, ad;
   char *a;

   trx = tree;
   t = bheader[trx].rootnode;
   if (t)    {
       read_node(t, &trnode);
       fnd = btreescan(&t, k, &a);
       ad = leaflevel(&t, &a, &i);
       if (i == trnode.keyct + 1)    {
           i = 0;
           t = trnode.rtsib;
       }
       currnode[trx] = t;
       currkno[trx] = i;
   }
   return fnd ? ad : 0;
}

/* ----------- Search tree ------------- */
static int btreescan(RPTR *t, char *k, char **a)
{
   int nl;
   do    {
       if (nodescan(k, a))    {
           while (compare_keys(*a, k) == FALSE)
               if (scanprev(t, a) == 0)
                   break;
           if (compare_keys(*a, k))
               scannext(t, a);
           return TRUE;
       }
       nl = trnode.nonleaf;
       if (nl)    {
```

(continued on page 144)

Listing Three *(Listing continued, text begins on page 121.)*

```
            *t = *((RPTR *) (*a - ADR));
            read_node(*t, &trnode);
        }
    }   while (nl);
    return FALSE;
}

/* ------------------ Search node ------------ */
static int nodescan(char *keyvalue, char **nodeadr)
{
    int i;
    int result;

    *nodeadr = trnode.keyspace;
    for (i = 0; i < trnode.keyct; i++)    {
        result = compare_keys(keyvalue, *nodeadr);
        if (result == FALSE)
            return TRUE;
        if (result < 0)
            return FALSE;
        *nodeadr += ENTLN;
    }
    return FALSE;
}

/* ------------- Compare keys ------------ */
static int compare_keys(char *a, char *b)
{
    int len = KLEN, cm;

    while (len--)
        if ((cm = (int) *a++ - (int) *b++) != 0)
            break;
    return cm;
}

/* ------------ Compute current file address  ------------ */
static RPTR fileaddr(RPTR t, char *a)
{
    RPTR cn, ti;
    int i;

    ti = t;
    cn = leaflevel(&ti, &a, &i);
    read_node(t, &trnode);
    return cn;
}

/* ---------------- Navigate down to leaf level ----------- */
static RPTR leaflevel(RPTR *t, char **a, int *p)
{
    if (trnode.nonleaf == FALSE)    { /* already at a leaf? */
        *p = (*a - trnode.keyspace) / ENTLN + 1;
        return *((RPTR *) (*a + KLEN));
    }
    *p = 0;
    *t = *((RPTR *) (*a + KLEN));
    read_node(*t, &trnode);
    *a = trnode.keyspace;
    while (trnode.nonleaf)    {
        *t = trnode.key0;
        read_node(*t, &trnode);
    }
    return trnode.key0;
}

/* -------------- Delete a key  ------------- */
int deletekey(int tree, char *x, RPTR ad)
{
    BTREE *qp, *yp;
    int rt_len, comb;
    RPTR p, adr, q, *b, y, z;
    char *a;

    trx = tree;
    if (trx >= MXTREES || fp[trx] == 0)
        return ERROR;
    p = bheader[trx].rootnode;
    if (p == 0)
        return OK;
    read_node(p, &trnode);
    if (btreescan(&p, x, &a) == FALSE)
        return OK;
    adr = fileaddr(p, a);
    while (adr != ad)    {
        adr = scannext(&p, &a);
        if (compare_keys(a, x))
            return OK;
    }
    if (trnode.nonleaf)    {
        b = (RPTR *) (a + KLEN);
        q = *b;
        if ((qp = malloc(NODE)) == NULL)
            memerr();
        read_node(q, qp);
        while (qp->nonleaf)    {
            q = qp->key0;
            read_node(q, qp);
        }
    /* Move the left-most key from the leaf
                        to where the deleted key is */
        memmove(a, qp->keyspace, KLEN);
        write_node(p, &trnode);
        p = q;
        trnode = *qp;
        a = trnode.keyspace;
        b = (RPTR *) (a + KLEN);
```

(continued on page 146)

Listing Three *(Listing continued, text begins on page 121.)*

```
    trnode.key0 = *b;
    free(qp);
}
currnode[trx] = p;
currkno[trx] = (a - trnode.keyspace) / ENTLN;
rt_len = (trnode.keyspace + (bheader[trx].m * ENTLN)) - a;
memmove(a, a+ENTLN, rt_len);
memset(a+rt_len, '\0', ENTLN);
trnode.keyct--;
if (currkno[trx] > trnode.keyct)    {
    if (trnode.rtsib)    {
        currnode[trx] = trnode.rtsib;
        currkno[trx] = 0;
    }
    else
        currkno[trx]--;
}
while (trnode.keyct <= bheader[trx].m / 2 &&
                    p != bheader[trx].rootnode)    {
    comb = FALSE;
    z = trnode.prntnode;
    if ((yp = malloc(NODE)) == NULL)
        memerr();
    if (trnode.rtsib)    {
        y = trnode.rtsib;
        read_node(y, yp);
        if (yp->keyct + trnode.keyct <
                bheader[trx].m && yp->prntnode == z)    {
            comb = TRUE;
            implode(&trnode, yp);
        }
    }
    if (comb == FALSE && trnode.lfsib)    {
        y = trnode.lfsib;
        read_node(y, yp);
        if (yp->prntnode == z)    {
            if (yp->keyct + trnode.keyct <
                                bheader[trx].m) {
                comb = TRUE;
                implode(yp, &trnode);
            }
            else    {
                redist(yp, &trnode);
                write_node(p, &trnode);
                write_node(y, yp);
                free(yp);
                return OK;
            }
        }
    }
    if (comb == FALSE)    {
        y = trnode.rtsib;
        read_node(y, yp);
        redist(&trnode, yp);
        write_node(y, yp);
        write_node(p, &trnode);
        free(yp);
        return OK;
    }

        free(yp);
        p = z;
        read_node(p, &trnode);
    }
    if (trnode.keyct == 0)    {
        bheader[trx].rootnode = trnode.key0;
        trnode.nonleaf = FALSE;
        trnode.key0 = 0;
        trnode.prntnode = bheader[trx].rlsed_node;
        bheader[trx].rlsed_node = p;
    }
    if (bheader[trx].rootnode == 0)
        bheader[trx].rightmost = bheader[trx].leftmost = 0;
    write_node(p, &trnode);
    return OK;
}

/* ------------ Combine two sibling nodes. ------------- */
static void implode(BTREE *left, BTREE *right)
{
    RPTR lf, rt, p;
    int rt_len, lf_len;
    char *a;
    RPTR *b;
    BTREE *par;
    RPTR c;
    char *j;

    lf = right->lfsib;
    rt = left->rtsib;
    p = left->prntnode;
    if ((par = malloc(NODE)) == NULL)
        memerr();
    j = childptr(lf, p, par);
    /* --- move key from parent to end of left sibling --- */
    lf_len = left->keyct * ENTLN;
    a = left->keyspace + lf_len;
    memmove(a, j, KLEN);
    memset(j, '\0', ENTLN);
    /* --- move keys from right sibling to left --- */
    b = (RPTR *) (a + KLEN);
    *b = right->key0;
    rt_len = right->keyct * ENTLN;
    a = (char *) (b + 1);
    memmove(a, right->keyspace, rt_len);
    /* --- point lower nodes to their new parent --- */
    if (left->nonleaf)
        adopt(b, right->keyct + 1, lf);
```

```
    /* --- if global key pointers -> to the right sibling,
                                    change to -> left --- */
    if (currnode[trx] == left->rtsib)    {
        currnode[trx] = right->lfsib;
        currkno[trx] += left->keyct + 1;
    }
    /* --- update control values in left sibling node --- */
    left->keyct += right->keyct + 1;
    c = bheader[trx].rlsed_node;
    bheader[trx].rlsed_node = left->rtsib;
    if (bheader[trx].rightmost == left->rtsib)
        bheader[trx].rightmost = right->lfsib;
    left->rtsib = right->rtsib;
    /* --- point the deleted node's right brother
                                to this left brother --- */
    if (left->rtsib)    {
        read_node(left->rtsib, right);
        right->lfsib = lf;
        write_node(left->rtsib, right);
    }
    memset(right, '\0', NODE);
    right->prntnode = c;
    /* --- remove key from parent node --- */
    par->keyct--;
    if (par->keyct == 0)
        left->prntnode = 0;
    else    {
        rt_len = par->keyspace + (par->keyct * ENTLN) - j;
        memmove(j, j+ENTLN, rt_len);
    }
    write_node(lf, left);
    write_node(rt, right);
    write_node(p, par);
    free(par);
}

/* ------------------ Insert key  ------------------- */
int insertkey(int tree, char *x, RPTR ad, int unique)
{
    char k[MXKEYLEN + 1], *a;
    BTREE *yp;
    BTREE *bp;
    int nl_flag, rt_len, j;
    RPTR t, p, sv;
    RPTR *b;
    int lshft, rshft;

    trx = tree;
    if (trx >= MXTREES || fp[trx] == 0)
        return ERROR;
    p = 0;
    sv = 0;
    nl_flag = 0;
    memmove(k, x, KLEN);
t = bheader[trx].rootnode;
/* --------------- Find insertion point ------- */
if (t)    {
    read_node(t, &trnode);
    if (btreescan(&t, k, &a))    {
        if (unique)
            return ERROR;
        else    {
            leaflevel(&t, &a, &j);
            currkno[trx] = j;
        }
    }
    else
        currkno[trx] = ((a - trnode.keyspace) / ENTLN)+1;
    currnode[trx] = t;
}
/* --------- Insert key into leaf node -------------- */
while (t)    {
    nl_flag = 1;
    rt_len = (trnode.keyspace+(bheader[trx].m*ENTLN))-a;
    memmove(a+ENTLN, a, rt_len);
    memmove(a, k, KLEN);
    b = (RPTR *) (a + KLEN);
    *b = ad;
    if (trnode.nonleaf == FALSE)    {
        currnode[trx] = t;
        currkno[trx] = ((a - trnode.keyspace) / ENTLN)+1;
    }
    trnode.keyct++;
    if (trnode.keyct <= bheader[trx].m)    {
        write_node(t, &trnode);
        return OK;
    }
    /* --- Redistribute keys between sibling nodes ---*/
    lshft = FALSE;
    rshft = FALSE;
    if ((yp = malloc(NODE)) == NULL)
        memerr();
    if (trnode.lfsib)    {
        read_node(trnode.lfsib, yp);
        if (yp->keyct < bheader[trx].m &&
                yp->prntnode == trnode.prntnode)    {
            lshft = TRUE;
            redist(yp, &trnode);
            write_node(trnode.lfsib, yp);
        }
    }
    if (lshft == FALSE && trnode.rtsib)    {
        read_node(trnode.rtsib, yp);
        if (yp->keyct < bheader[trx].m &&
                yp->prntnode == trnode.prntnode)    {
            rshft = TRUE;
            redist(&trnode, yp);
            write_node(trnode.rtsib, yp);
        }
    }
    free(yp);
    if (lshft || rshft)    {
        write_node(t, &trnode);
        return OK;
    }
    p = nextnode();
    /* ---------- Split node -------------------- */
    if ((bp = malloc(NODE)) == NULL)
        memerr();
    memset(bp, '\0', NODE);
    trnode.keyct = (bheader[trx].m + 1) / 2;
    b = (RPTR *)
          (trnode.keyspace+((trnode.keyct+1)*ENTLN)-ADR);
    bp->key0 = *b;
    bp->keyct = bheader[trx].m - trnode.keyct;
    rt_len = bp->keyct * ENTLN;
    a = (char *) (b + 1);
    memmove(bp->keyspace, a, rt_len);
    bp->rtsib = trnode.rtsib;
    trnode.rtsib = p;
    bp->lfsib = t;
    bp->nonleaf = trnode.nonleaf;
    a -= ENTLN;
    memmove(k, a, KLEN);
    memset(a, '\0', rt_len+ENTLN);
    if (bheader[trx].rightmost == t)
        bheader[trx].rightmost = p;
    if (t == currnode[trx] &&
                currkno[trx]>trnode.keyct)    {
        currnode[trx] = p;
        currkno[trx] -= trnode.keyct + 1;
    }
    ad = p;
    sv = t;
    t = trnode.prntnode;
    if (t)
        bp->prntnode = t;
    else    {
        p = nextnode();
        trnode.prntnode = p;
        bp->prntnode = p;
    }
    write_node(ad, bp);
    if (bp->rtsib)    {
        if ((yp = malloc(NODE)) == NULL)
            memerr();
            read_node(bp->rtsib, yp);
            yp->lfsib = ad;
            write_node(bp->rtsib, yp);
            free(yp);
        }
        if (bp->nonleaf)
            adopt(&bp->key0, bp->keyct + 1, ad);
        write_node(sv, &trnode);
        if (t)    {
            read_node(t, &trnode);
            a = trnode.keyspace;
            b = &trnode.key0;
            while (*b != bp->lfsib)    {
                a += ENTLN;
                b = (RPTR *) (a - ADR);
            }
        }
        free(bp);
    }
    /* ---------------------- new root --------------------- */
    if (p == 0)
        p = nextnode();
    if ((bp = malloc(NODE)) == NULL)
        memerr();
    memset(bp, '\0', NODE);
    bp->nonleaf = nl_flag;
    bp->prntnode = 0;
    bp->rtsib = 0;
    bp->lfsib = 0;
    bp->keyct = 1;
    bp->key0 = sv;
    *((RPTR *) (bp->keyspace + KLEN)) = ad;
    memmove(bp->keyspace, k, KLEN);
    write_node(p, bp);
    free(bp);
    bheader[trx].rootnode = p;
    if (nl_flag == FALSE)    {
        bheader[trx].rightmost = p;
        bheader[trx].leftmost = p;
        currnode[trx] = p;
        currkno[trx] = 1;
    }
    return OK;
}

/* ----- redistribute keys in sibling nodes ------ */
static void redist(BTREE *left, BTREE *right)
{
    int n1, n2, len;
    RPTR z;
    char *c, *d, *e;
    BTREE *zp;

    n1 = (left->keyct + right->keyct) / 2;
    if (n1 == left->keyct)
        return;
    n2 = (left->keyct + right->keyct) - n1;
    z = left->prntnode;
    if ((zp = malloc(NODE)) == NULL)
        memerr();
    c = childptr(right->lfsib, z, zp);
```

(continued on page 148)

Listing Three *(Listing continued, text begins on page 121.)*

```
    if (left->keyct < right->keyct)    {
        d = left->keyspace + (left->keyct * ENTLN);
        memmove(d, c, KLEN);
        d += KLEN;
        e = right->keyspace - ADR;
        len = ((right->keyct - n2 - 1) * ENTLN) + ADR;
        memmove(d, e, len);
        if (left->nonleaf)
            adopt(d, right->keyct - n2, right->lfsib);
        e += len;
        memmove(c, e, KLEN);
        e += KLEN;
        d = right->keyspace - ADR;
        len = (n2 * ENTLN) + ADR;
        memmove(d, e, len);
        memset(d+len, '\0', e-d);
        if (right->nonleaf == 0 &&
                        left->rtsib == currnode[trx])
            if (currkno[trx] < right->keyct - n2)    {
                currnode[trx] = right->lfsib;
                currkno[trx] += n1 + 1;
            }
            else
                currkno[trx] -= right->keyct - n2;
        }
    else    {
        e = right->keyspace+((n2-right->keyct)*ENTLN)-ADR;
        memmove(e, right->keyspace-ADR,
            (right->keyct * ENTLN) + ADR);
        e -= KLEN;
        memmove(e, c, KLEN);
        d = left->keyspace + (n1 * ENTLN);
        memmove(c, d, KLEN);
        memset(d, '\0', KLEN);
        d += KLEN;
        len = ((left->keyct - n1 - 1) * ENTLN) + ADR;
        memmove(right->keyspace-ADR, d, len);
        memset(d, '\0', len);
        if (right->nonleaf)
            adopt(right->keyspace - ADR,
                        left->keyct - n1, left->rtsib);
        if (left->nonleaf == FALSE)
            if (right->lfsib == currnode[trx] &&
                                currkno[trx] > n1)    {
                currnode[trx] = left->rtsib;
                currkno[trx] -= n1 + 1;
            }
            else if (left->rtsib == currnode[trx])
                currkno[trx] += left->keyct - n1;
    }
    right->keyct = n2;
    left ->keyct = n1;
    write_node(z, zp);
    free(zp);
}

/* ---------- assign new parents to child nodes ---------- */
static void adopt(void *ad, int kct, RPTR newp)
{
    char *cp;
    BTREE *tmp;

    if ((tmp = malloc(NODE)) == NULL)
        memerr();
    while (kct--)    {
        read_node(*(RPTR *)ad, tmp);
        tmp->prntnode = newp;
        write_node(*(RPTR *)ad, tmp);
        cp = ad;
        cp += ENTLN;
        ad = cp;
    }
    free(tmp);
}

/* ----- compute node address for a new node -----*/
static RPTR nextnode(void)
{
    RPTR p;
    BTREE *nb;

    if (bheader[trx].rlsed_node)    {
        if ((nb = malloc(NODE)) == NULL)
            memerr();
        p = bheader[trx].rlsed_node;
        read_node(p, nb);
        bheader[trx].rlsed_node = nb->prntnode;
        free(nb);
    }
    else
        p = bheader[trx].endnode++;
    return p;
}

/* ----- next sequential key ------- */
RPTR nextkey(int tree)
{
    trx = tree;
    if (currnode[trx] == 0)
        return firstkey(trx);
    read_node(currnode[trx], &trnode);
    if (currkno[trx] == trnode.keyct)    {
        if (trnode.rtsib == 0)    {
            return 0;
        }
        currnode[trx] = trnode.rtsib;
        currkno[trx] = 0;
        read_node(trnode.rtsib, &trnode);
    }
    else
        currkno[trx]++;
    return *((RPTR *)
            (trnode.keyspace+(currkno[trx]*ENTLN)-ADR));
}

/* ---------- previous sequential key ---------- */
RPTR prevkey(int tree)
{
    trx = tree;
    if (currnode[trx] == 0)
        return lastkey(trx);
    read_node(currnode[trx], &trnode);
    if (currkno[trx] == 0)    {
        if (trnode.lfsib == 0)
            return 0;
        currnode[trx] = trnode.lfsib;
        read_node(trnode.lfsib, &trnode);
        currkno[trx] = trnode.keyct;
    }
    else
        currkno[trx]--;
    return *((RPTR *)
        (trnode.keyspace + (currkno[trx] * ENTLN) - ADR));
}

/* ------------ first key ------------ */
RPTR firstkey(int tree)
{
    trx = tree;
    if (bheader[trx].leftmost == 0)
        return 0;
    read_node(bheader[trx].leftmost, &trnode);
    currnode[trx] = bheader[trx].leftmost;
    currkno[trx] = 1;
    return *((RPTR *) (trnode.keyspace + KLEN));
}

/* ------------ last key ----------------- */
RPTR lastkey(int tree)
{
    trx = tree;
    if (bheader[trx].rightmost == 0)
        return 0;
    read_node(bheader[trx].rightmost, &trnode);
    currnode[trx] = bheader[trx].rightmost;
    currkno[trx] = trnode.keyct;
    return *((RPTR *)
        (trnode.keyspace + (trnode.keyct * ENTLN) - ADR));
}

/* -------- scan to the next sequential key ------ */
static RPTR scannext(RPTR *p, char **a)
{
    RPTR cn;

    if (trnode.nonleaf)    {
        *p = *((RPTR *) (*a + KLEN));
        read_node(*p, &trnode);
        while (trnode.nonleaf)    {
            *p = trnode.key0;
            read_node(*p, &trnode);
        }
        *a = trnode.keyspace;
        return *((RPTR *) (*a + KLEN));
    }
    *a += ENTLN;
    while (-1)    {
        if ((trnode.keyspace + (trnode.keyct)
                * ENTLN) != *a)
            return fileaddr(*p, *a);
        if (trnode.prntnode == 0 || trnode.rtsib == 0)
            break;
        cn = *p;
        *p = trnode.prntnode;
        read_node(*p, &trnode);
        *a = trnode.keyspace;
        while (*((RPTR *) (*a - ADR)) != cn)
            *a += ENTLN;
    }
    return 0;
}

/* ---- scan to the previous sequential key ---- */
static RPTR scanprev(RPTR *p, char **a)
{
    RPTR cn;

    if (trnode.nonleaf)    {
        *p = *((RPTR *) (*a - ADR));
        read_node(*p, &trnode);
        while (trnode.nonleaf)    {
            *p = *((RPTR *)
                (trnode.keyspace+(trnode.keyct)*ENTLN-ADR));
            read_node(*p, &trnode);
        }
        *a = trnode.keyspace + (trnode.keyct - 1) * ENTLN;
        return *((RPTR *) (*a + KLEN));
    }
    while (-1)    {
        if (trnode.keyspace != *a)    {
            *a -= ENTLN;
            return fileaddr(*p, *a);
        }
        if (trnode.prntnode == 0 || trnode.lfsib == 0)
            break;
        cn = *p;
        *p = trnode.prntnode;
```

(continued on page 150)

Listing Three *(Listing continued, text begins on page 121.)*

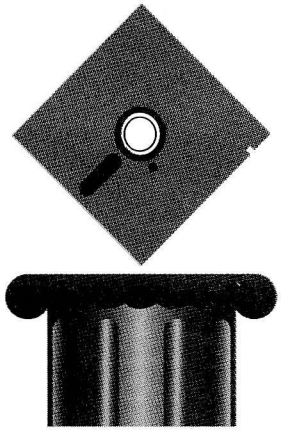
```
        read_node(*p, &trnode);
        *a = trnode.keyspace;
        while (*((RPTR *) (*a - ADR)) != cn)
            *a += ENTLN;
    }
    return 0;
}

/* ------ locate pointer to child ---- */
static char *childptr(RPTR left, RPTR parent, BTREE *btp)
{
    char *c;

    read_node(parent, btp);
    c = btp->keyspace;
    while (*((RPTR *) (c - ADR)) != left)
        c += ENTLN;
    return c;
}

/* -------------- current key value ---------- */
void keyval(int tree, char *ky)
{
    RPTR b, p;
    char *k;
    int i;

    trx = tree;
    b = currnode[trx];
    if (b)    {
        read_node(b, &trnode);
        i = currkno[trx];
        k = trnode.keyspace + ((i - 1) * ENTLN);
        while (i == 0)    {
            p = b;
            b = trnode.prntnode;
            read_node(b, &trnode);
            for (; i <= trnode.keyct; i++)    {
                k = trnode.keyspace + ((i - 1) * ENTLN);
                if (*((RPTR *) (k + KLEN)) == p)
                    break;
            }
        }
        memmove(ky, k, KLEN);
    }
}

/* --------------- current key ---------- */
RPTR currkey(int tree)
{
    RPTR f = 0;

    trx = tree;
    if (currnode[trx])    {
        read_node(currnode[trx], &trnode);
        f = *( (RPTR *)
            (trnode.keyspace+(currkno[trx]*ENTLN)-ADR));
    }
    return f;
}

/* ---------- read a btree node ---------- */
static void read_node(RPTR nd, void *bf)
{
    bseek(nd);
    fread(bf, NODE, 1, fp[trx]);
    if (ferror(fp[trx]))    {
        errno = D_IOERR;
        dberror();
    }
}

/* ---------- write a btree node ---------- */
static void write_node(RPTR nd, void *bf)
{
    bseek(nd);
    fwrite(bf, NODE, 1, fp[trx]);
    if (ferror(fp[trx]))    {
        errno = D_IOERR;
        dberror();
    }
}

/* ---------- seek to the b-tree node ---------- */
static void bseek(RPTR nd)
{
    if (fseek(fp[trx],
          (long) (NODE+((nd-1)*NODE)), SEEK_SET) == ERROR) {
        errno = D_IOERR;
        dberror();
    }
}

/* ---------- out of memory error ---------- */
static void memerr(void)
{
    errno = D_OM;
    dberror();
}
```

End Listings

Listing One *(Text begins on page 131.)*

```
{-------------------------------------------------------}
{                                                       }
{                        JXDESK                         }
{                                                       }
{  Jeff's Experimental Desktop Manager for Turbo Vision }
{                                                       }
{                              by Jeff Duntemann        }
{                              For Turbo Pascal V6.0    }
{                                                       }
{-------------------------------------------------------}

PROGRAM JDesk;

{ These are all Turbo Vision units: }
USES Objects, Drivers, Views, Menus, Dialogs, App;


CONST
  SysStatCmd = 102;
  NullCmd    = 101;


TYPE
  PDesk = ^TDesk;
  TDesk = OBJECT(TApplication)
    PROCEDURE HandleEvent(VAR Event: TEvent); VIRTUAL;
    PROCEDURE InitMenuBar; VIRTUAL;
    PROCEDURE InitStatusLine; VIRTUAL;
    PROCEDURE SystemStatistics;
  END;


VAR
  Desk: TDesk;   { Allocate an instantiation of TDesk }

{ TDesk method definitions: }


PROCEDURE TDesk.HandleEvent(VAR Event: TEvent);


BEGIN
  TApplication.HandleEvent(Event);
  IF Event.What = evCommand THEN  { If the event was a command }
  BEGIN
    CASE Event.Command of
      { The system invokes a method in response to a command: }
      SysStatCmd : SystemStatistics;
    ELSE
      Exit;  { Exit the event handler; i.e., do nothing }
    END;
    ClearEvent(Event);
  END;
END;


PROCEDURE TDesk.SystemStatistics;


VAR
  R: TRect;
  D: PDialog;
  C: Word;


BEGIN
  { Create a new dialog: }
  R.Assign(25, 5, 55, 14);
  D := New(PDialog,Init(R,'System Stats'));

  { Create and insert controls into the dialog: }
  R.Assign(9, 6, 21, 8);
  D^.Insert(New(PButton,Init(R,'OK',cmCancel,bfNormal)));

  { Execute the modal dialog: }
  C := DeskTop^.ExecView(D);
END;


PROCEDURE TDesk.InitMenuBar;


VAR
  R: TRect;


BEGIN
  GetExtent(R);
  R.B.Y := R.A.Y+1;
  MenuBar := New(PMenuBar,Init(R,NewMenu(
    NewSubMenu('~p~',hcNoContext,NewMenu(
      NewItem('~A~bout',          '',0,NullCmd,hcNoContext,
      NewItem('How to ~R~egister','',0,NullCmd,hcNoContext,
      nil))),
    NewSubMenu('~S~ystem', hcNoContext, NewMenu(
      NewItem('~S~tatistics',    '',0,SysStatCmd,hcNoContext,
      NewItem('Set ~T~ime',      '',0,NullCmd,hcNoContext,
      NewItem('Set ~D~ate',      '',0,NullCmd,hcNoContext,
      NewItem('~R~un DOS app...','',0,NullCmd,hcNoContext,
      NewLine(
      NewItem('E~x~it','Alt-X',kbAltX,cmQuit,hcNoContext,
      nil)))))),
    NewSubMenu('Address ~B~ook',hcNoContext,NewMenu(
      NewItem('~O~pen book',  '',0,NullCmd,hcNoContext,
      NewItem('~C~reate book','',0,NullCmd,hcNoContext,
      NewItem('~P~rint book', '',0,NullCmd,hcNoContext,
      nil)))),
    NewSubMenu('~T~erm',hcNoContext,NewMenu(
      NewItem('Link to ~M~CI',        '',0,NullCmd,hcNoContext,
      NewItem('Link to ~C~ompuServe','',0,NullCmd,hcNoContext,
      NewItem('Link to ~B~ix',        '',0,NullCmd,hcNoContext,
      NewItem('~T~erminal window',    '',0,NullCmd,hcNoContext,
      nil))))),
  nil)))))));
END;


PROCEDURE TDesk.InitStatusLine;

VAR
  R: TRect;


BEGIN
  GetExtent(R);
  R.A.Y := R.B.Y-1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-X~ Exit',kbAltX,cmQuit,nil),nil)));
END;


BEGIN
  Desk.Init;
  Desk.Run;
  Desk.Done;
END.
```

End Listing



Third-party CAD/CAM developers may be interested in the CAD/CAM Developer's Kit (CCDK) from **Building Block Software**. CCDK is a toolbox of C functions that provides full 3-D DXF, 2-D and 3-D display and geometry operations, and list-management capabilities.

DDJ spoke with Kevin Green of Karta Technology, who said that "We needed to create a drawing with AutoCAD and bring the drawing into our program, which links to databases or a variety of other things. The big advantage [of CCDK] was the DXF support. It would have taken us a lot of time to develop this on our own." Kevin also mentioned that the documentation and tech support are exceptional.

CCDK can be used for developing DXF file viewer add-ons to database programs, numerical code generators, finite element mesh generators, 3-D piping layout tools, and printed circuit board layout checking programs. CCDK augments Autodesk's C-binding environment with a full set of functions for CAD/CAM operations, saving developers the task of writing all the computational functions themselves. Object hierarchies and data encapsulation make programming faster and minimize the number of public routines required to support all of CCDK's operations. Full DXF Release 10 functionality is supported, so programs can read and write both ASCII and binary DXF formats. Block inserts can be exploded into geometry and solid models can be generated for shading with AutoSolid. The routines are adaptable to popular graphics libraries. CCDK is compatible with Microsoft C 5.1 and 6.0 and Borland Turbo C 2.0, and supports Metaware and Watcom compilers as well. Single programmer licenses start at $1,295. Reader service no. 20.

Building Block Software Inc.
P.O. Box 1373
Somerville, MA 02144
617-628-5217

The Am29050 Floating Point Processor is a general-purpose, 32-bit microprocessor from **Advanced Micro Devices**. Featuring a RISC architecture, the Am29050 has a pipelined, on-chip floating point unit that performs IEEE-compatible, single- and double-precision arithmetic at a rate of 80 MFLOPS at 40 MHz. Other features improve the performance of loads and branches, allowing sustained integer performance of 32 MIPS at 40 MHz.

The Am 29050 microprocessor is pin and code-compatible with the Am 29000 microprocessor, so existing Am29000 microprocessor applications can use the Am29050 processor without modification and can achieve a factor-of-four increase in performance for such floating-point-intensive applications as graphics (faster 3-D performance) and laser printers (highest page description language performance). Reader service no. 21.

Advanced Micro Devices
901 Thompson Pl.
Sunnyvale, CA 94088
408-732-2400

The ToolBook Author's Resource Kit (ARK) is now available from **Asymetrix**. The ARK includes a license to distribute a royalty-free runtime version of ToolBook and three DLLs, as well as development tools. ToolBook 1.0 is a software construction set for building graphical applications without using traditional programming languages. You can use ToolBook to build Windows 3.0 applications.

ARK includes two development utilities, and two online applications offer ideas on user interface design for ToolBook applications. The Windows DLLs include a DLL that provides access to and file manipulation of dBase III data files; another for determining the characteristics of a display device, for determining which fonts are available for display and printing, and for setting and retrieving values in a Windows initialization file directly from a ToolBook script; and one for copying and deleting DOS files and for performing other DOS functions within a ToolBook script.

Merillin Paris, head engineer on the project, told *DDJ* that "a misconception some people have is that ToolBook is interpreted. It is not interpreted, it is p-code. You can strip the code down to binary and remove the script to protect your source code." The utility included for that is Script Remover. Another utility included is BookLook, for examining and changing an object's properties. The ARK is priced at $450, and for another $495 you can subscribe to the Asymetrix Developer Support Partnership program and be assigned to a support engineer for one year of technical support. Reader service no. 22.

Asymetrix Corp.
110 110th Ave. NE, Ste. 717
Bellevue, WA 98004
206-637-1500

If you need high-performance, interrupt-driven control of asynchronous communications in your C programs, you might want to check out the new SilverComm C Async Library from **SilverWare**. The library is customizable, as full source code is included.

A device event monitor capability permits execution of your C functions at interrupt time. These functions can set flags that signal an event in your application. And you can create functions that process the interrupt.

Jim McCusker of Electronics Unlimited has been a beta tester for all evolutions of the Async package. He told *DDJ*, "We have used the library on various projects here. We do a lot of custom software and hardware for the process-control industry. This package takes the burden off the user — you don't need to know about serial ports on a PC. You can write a comm program in about 10 minutes for a dumb terminal. You can grow with this package — it has extra functions, such as timers for setting a delay and basic video I/O. And it supports multiple communication ports simultaneously, asynchronously. One of the best things is the manual; each function is described in detail and has an example of its use."

The SilverComm C Async Library features huge model queues; terminal emulation; XModem, YModem, YModem batch, and ASCII; background timers; enhanced compatibility; flow control; smart modem support; high-level remote input; character filtering; and supports up to 115K baud. The library sells for $259. Reader service no. 25.

SilverWare Inc.
3010 LBJ Freeway, Ste. 740
Dallas, TX 75234
214-247-0131

UniFLEX.wks, from **UniFLEX Computing**, is a real-time cross-development package for VME designers. Previously only available as a self-hosted development system, UniFLEX is now fully integrated with Sun Microsystems SPARC and 68000 workstations. UniFLEX is Unix-compatible, multitasking, and multiprocessing. You can use Sun's Unix development tools to edit, track, and manage system software, and then use UniFLEX.wks to compile, link, and down-

(continued on page 157)

(continued from page 153)
load object modules over the Ethernet to the target VME system, and then debug with UniFLEX tools such as the GBD symbolic debugger.

UniFLEX.wks produces a pseudo-disk that contains the UniFLEX kernel and application programs and which can be placed in ROM for embedded system applications. Real-time features include programmer-declared real-time priorities, task synchronization, contiguous file system, and intertask communication using binary semaphores or named message exchanges. Tightly coupled multiprocessing allows multiple CPUs on a single backplane executing in concert with dynamic load balancing. UniFLEX.wks allows for the development of X Window (with Motif) applications on target systems using Sun Workstations. Reader service no. 23.
UniFLEX Computing Ltd.
111 Providence Rd.
Chapel Hill, NC 27514
919-493-1451; 800-486-1000

The C++ Rider package for source code analysis and online help for C and C++ is now available from **Western Wares**. Based on the CC-Rider tool for C, C++ Rider consists of a pop-up source-browsing utility (CCRIDER) and a source code analyzer (CCSYM). CCRIDER can be used with any text-mode editor, integrated environment, or debugger to provide hypertext symbol referencing as you program. You can display, view source, edit, or paste any symbol definition using a hotkey, and you can recall all classes, structures, unions, enums, enum values, typedefs, macros, functions, and data symbols.

CCSYM performs analysis and symbol database management for C and C++ programs, and can generate function prototype header files as required by the C++ language. The package supports Microsoft C 6.0, Zortech C++ 2.1, and Borland's Turbo C++. It is available in both a standard edition and a professional edition. Contact the company for pricing. Reader service no. 26.
Western Wares
Box C
Norwood, CO 81423
303-327-4898

Macintosh programmers now have an alternative to TextEdit with the release of **DataPak Software's** Word Solution Engine, a new module for basic text editing with multiple font and style support. The company claims that Word Solution Engine has superior buffering capabilities (16 Mbytes) to TextEdit (which has a 32K limit), is faster, and formats better.

Ben Cahan of MacToolkit concurs. He told *DDJ* that Word Solution Engine "is very complete and works flawlessly. I'm working in an application that is text-editing intensive, and this product handles pretty much everything. It could save one or two years of work writing your own text editor. This is the only solution I've seen so far for quality text editing."

Word Solution Engine features fast editing and display of large text files, Script Manager support, Quickdraw color support, and "User Styles." Also multiple stylations, tab handling, I/O and printing support, "scrap" support, and full-justification. The package includes a programmer's guide, sample source code that demonstrates each function, interface files for MPW Pascal and Think C, and linkable object files for both. Object code is being offered at an introductory, one-time royalty-free rate of $300. Reader service no. 24.
DataPak Software Inc.
9317 NE Highway 99, Ste. G
Vancouver, WA 98665-8900
206-573-9155
800-327-6703

The Solid Link Library from **Solid Software** is a telecommunications package for C, C++, and Modula-2 programmers. It includes the difficult-to-implement ZModem protocol, as well as XModem, XModem-1K, YModem, Modem7, TeLink, and SEAlink file transfer protocols with concurrent and background transfers. XON/XOFF, DSR/DTR, CTS, RTS, as well as Ctrl-C/Crtl-K handshaking are supported. The package includes a sample terminal program with source code. Supports most popular C, C++, and Modula-2 compilers. Solid Link for C and Modula-2 is $199, with source $498; add $50 for C++. Reader service no. 27.
Solid Software
227 West Fourth St.
Cincinnati, OH 45202-2645
513-651-1133

Microway has announced NDP C++, a full 32-bit, protected-mode C++ compiler that is AT&T Release 2.0-compatible. It runs on the 80386, 80486, and i860, under DOS, Unix, Xenix, and SunOS. NDP C++ utilizes the same code generators and global optimizing technologies used in the NDP Fortran, C, and Pascal compilers, and includes extensions that specifically handle C++ features, such as function inlining, classes, constructors and destructors, single and multiple inheritance, and function and operator overloading. Microway integrated C++ extensions into their ANSI C front end and extended their optimizer and code generator to handle C++-specific operations. They claim this approach increases the execution speed of the compiler and improves the quality of the generated code. The 80386 version is $895, the 80486 version is $1,195, and the i860 version is $1,995. All include one year of free updates. Reader service no. 28.
Microway
P.O. Box 79
Kingston, MA 02364
508-746-7341

The Multimedia Expo, held in San Francisco in early October, consisted of seminars, workshops, corporate presentations, and exhibits in the field of multimedia, the components of which are based on digital technology. Seminar topics included telecommunications and networking, desktop video workstations, voice command, multimedia engines and systems, CD-ROM XA production, full-motion video hardware and software, hypertext and hypermedia authoring, visualization, animation, artificial reality, and multimedia projects in education. The professional workshops covered similar topics, but included hands-on instruction and in-depth information. The thrust of the Expo was on the business applications of multimedia for corporate presentations and information systems. Sponsored by **American Expositions**, Multimedia Expo will be held in New York in May 1991 and again in San Francisco in October 1991. Call 212-226-4141 for information.

Communications Formulas and Algorithms by C. Britton Rorabaugh has been published by **McGraw-Hill**. Rorabaugh focuses on definitions, formulas, algorithms, and design data needed by communications engineers, programmers, and technicians. The author presents techniques for analyzing, simulating, designing, and testing communications systems. Each procedure, method, and algorithm includes the relevant mathematical notation and pseudocode, guidelines for selection and use, function plots, graphs, and diagrams, and practical examples. Topics range from probability distributions in communications to pulse modulation and transmission, from random processes to signals and spectra, and from communication channels to phase shift keying. The book sells for $39.95. ISBN: 0-07-053644-9. Reader service no. 29.
McGraw-Hill Publishing Co.
11 West 19th St.
New York, NY 10011
212-337-5945
800-262-4729

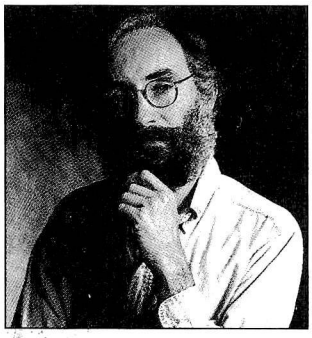
Help! is a help system for creating pop-up help and menus, from **Zaron Software**. Help! can be adapted to any application you buy or write yourself. Help! can be used with standard text files to create a custom "flat" help system. It automatically displays the correct help file for the program you are running, returns values or commands to the executing program or to DOS, provides context-sensitive help at every point in each program's operation, and can be used as a menu system as well. Sample help files and programs are included, as is a help system for commonly used DOS commands. Help! is a TSR, requires less than 14K of memory, and includes mouse support. The price is $49.95. Reader service no. 30.
Zaron Software
13100 Dulaney Valley Rd.
Glen Arm, MD 21057
301-592-3334

Objectworks\Smalltalk, Release 4, is a new generation object-oriented programming system from **ParcPlace Systems**. It is designed for creating customized, true color, graphic applications for heterogeneous computing environments under standard windowing systems. Other new features include the Smalltalk Portable Imaging Model (SPIM), incremental garbage collection, and support for international applications. Objectworks\Smalltalk is designed for creating interactive business and technical applications such as information management and decision analysis, systems design and simulation, and application-specific CASE tools. Objectworks\Smalltalk sells for $3,500. Reader service no. 32.
ParcPlace Systems
1550 Plymouth St.
Mountain View, CA 94043
415-691-6700

C++/Views from **CNS** is fully validated with Zortech C++ Version 2.1. C++/Views is an application development framework for Windows 3.0 development that provides object classes and productivity tools. C++/Views comes with over 60 ready-made C++ object classes that encapsulate the functionality within Windows. C++/Views also comes with a C++ object class browser, which manages such mechanics as building make files. An interface generator is also included for building C++ dialog objects. C++/Views retails for $495. Reader service no. 36.
CNS Inc.
Software Products Dept.
7090 Shady Oak Rd.;
Minneapolis, MN 55344
612-944-0170

DDJ

A Look at the Latest Videos



In these latter days of the 20th century, the format of choice for communicating with Americans is VCR, and if the computer industry knows anything, it knows formats. Today, software documentation, press releases, and memoirs of computer scientists are all making the move to video. The latest collection arrived just in time for the holidays. Here are the best of the lot.

The Dutch Boy's Finger: Edsgar Dijkstra in Concert. Rijksfilms, 90 min. The King of the Dutch Computer Science Insult Comics takes a few pokes at some well-chosen targets and a North Sea of laughs comes pouring in. Among the pokees: Americans (too anti-intellectual); IBM (too big); software engineering (the 20th-century mania for cooperation in everything); universities (their degeneration into graduate factories a threat to civilization); programming languages (Fortran is an infantile disorder, PL/I a fatal disease, Basic leaves students mentally mutilated beyond hope of regeneration, APL creates a new generation of coding bums, *natural language* programming is doomed to fail, and it's unscientific and a symptom of professional immaturity to call any of them *languages* at all); and John von Neumann (as the villain who introduced anthropomorphism into computer science). Based on his popular treatise, *Selected Writings on Computing: A Personal Perspective*, originally distributed in copier-paper format. R, strong language, adult themes.

Groupthink: Getting Ready for Groupware, CoLabVideo, 104 min. A collection of lectures, interviews, and case studies on the Groupthink model and its implementation. Michael Shrage and a group of Lotus groupware specialists discuss how all great discoveries are really collaborative efforts, a group of Apple Computer Area Associates examines the ways in which workgroup computing has helped Apple retain its technical edge, and, in an interview with a group of *Newsweek* reporters, Alvin Toffler explains how to detect and suppress individual thinking in groups. The second half of the video deals with legal and technical issues: A legal firm discusses the subtleties of the shrinkware group-license: If any of the group breaks the seal, is the entire group obligated by the terms and conditions of the contract? To its credit, the video frankly acknowledges some problems yet to be resolved in groupware; in one case study at a university function, a group of philosophy professors is shown failing to get anything to eat due to a nondeterminancy in the groupware algorithm. G, suitable for all viewers.

The Face Against the Glass. Loving Grace Cybernetics, 120 min. A challenging video, narrated by Mitch Kapor. Based on a short story by Franz Kafka, the video explores the consequences for human interface design of the fact that people are more adaptable than machines. Kafka himself wrote the best summation of the story: *Leopards break into the temple and drink up the sacrificial wine; this is repeated over and over again; eventually it becomes predictable, and is incorporated into the ceremony.* Many questions are left unresolved in this dark, troubling video: Is there any sequence of actions that will not come to seem to users the normal way to get a task accomplished, if they repeat it often enough? Is human interface design an art, a science, or religious ritual? NC-17, ritual acts, human sacrifice, desecration of icons.

The Apple/Claris HyperCard Announcement, Apple Computer Multimedia Lab, 38 min. A taut, white-knuckle suspense story with a Peckinpah-like burst of violence at the climax. After three years, Apple releases a new version of HyperCard. The tension is tangible, and the denouement is predictable when the company announces that the product, formerly bundled with all Macintoshes, will henceforth be a *competitively-priced* Claris product. A thriller. R, strong language, graphic violence.

A Tribute to Bill Gates, Microsoft Press Video Division, 3 min. All of Bill Gates's industry friends get together to express their appreciation and fondness for the great man. G, suitable for all viewers.

Our Best Marketing Secrets, Tandy/Radio Shack Videos, ?? min. This video could not be reviewed, as the copy received was blank.

Michael Swaine

Michael Swaine
editor-at-large

Index

3-D graphic, 857
3-D, 880
3, 915

16-bit accesses, to 8-bit adaptors, 429
16-bit operations, myth of, 433
16-bit VGA, 428,
    performance, 428
16-color, 589

32-bit 8086, 442
32-bit MS-DOS, 145

48-bit, 445

256-color, extended, 590

386, 613

1989 DDJ Index, 324-327

8080, 212
8086, 212
8086, 613

8088, 79
68000, overview, 216

"68040 Programming," 216-219

80286, 613

"80386 Debugging Features," 222-223
80386, 441, 523, 613, 678, 895
    debugging, 1019
    exceptions, 1020

A

"Algebraic Codes for Error Detection and Correction," 1112
"Are the Emperor's New Clothes Object-Oriented?," reference to, 20
"Assembly Language Lives!," 205-211
"Assembly Language Tricks of the Trade, 212-215
"Assembly Language Tricks of the Trade," reference to, 393
"AWK as a C Code Generator," 689-694
Above DISC Version 3.1, 479
Above Software, 479
Abrash, Michael, 205
Actor 2.0, 386, 3.0, 1026
Acumen Software, 1089
Addison-Wesley, 388, 460, 480
Adobe, 603
Advanced MS-DOS, 279
Aho, Alfred, 689
AI, 92, 695
Albrash, Michael, 428
algorithm, 964, 976
    B-tree, 1158
    Data Encryption Standard, 1067
    garbage collection, 396
    hashing, 176
    heuristic, 127
    mark-and-sweep, 397
    modern, 398
    pixel-ordering, 511
    sequential search, 1116
Algorithmics, 460
Allegro Common Lisp, 695
Allen Systems, 103
Allen, Norton T., 511
Almanac, 578
American Expositions, 1181
analyzer, digital differential, 593
AND, 1095
Anderson, Brian R., 924, 823
Andrews Data Systems, 103
animation structures, 24
animation, 21
ANSI Compiler, 679
ANSI preprocessor ###, 959
ANSI standard C, 468
Ansi C, 655, token pasting, 655
ANSI, 91
AnSoft Inc., 983
Apple Computer, 387, 649, 1108
Applied Microsystems Corp., 886
Applied Research, 1088
artificial neurons, 307
ASM, 243

aspect ration, 595
assembler specific features, 244
assembler, 217, 813
assembly language, 232, 243, 585
assert macros, 1001
Asymetrix's ToolBook, 968
Asymetrix, 1156, 1179
AT-But, 431
ATLast! Software, 103
attribute bits, 443
Auping, J.V., 795
AvCase, 480
Avocet, 480
AWK, for DOS, 693
Ayers, Kenneth E., 1082
Ayers, Kenneth, E., 435

B

"Bidirectional Associative Memory Systems in C++," 295-300
"Bounding Box Data Compression," 319-323
"Building A Hypertext System," 492-497
"Building an Efficient Help System," 501-505
B-Tran Basic to C translator, 104
B-tree, 560, 1158
    searching, 1159
    adding a key, 1160
    deleting a key, 1160
    code, 1160
background processes, 1097
Baldwin, Wahhab, 689
BAM, encoding, 295
    decoding, 295
    adoptive, 296
    systems, 296
    implementation, 298
Baran's Tech Letter, 983
Basic function, 809
Bauer, Barr E., 1097
benchmarks, 240
Bergman, Noel, 139
Better-C, 480
Bézier Curves, 603, 1096
    rendering, 604
Bézier equation, 604
Bézier, Pierre, 603
BGI, 1073
    with Turbo Pascal 4.0, 1073
bicubic splines, 695-703
bidirectional associative memory (BAM), 295
binary search trees, 131
binary tree, 175
bindery, 769
Bitson equation, 995
Blum, Adam, 295
Bolland, Jim, 416
Booch, Grady, 1075
Booleans, 771
Borland International, 285, 386, 464, 653, 669, 765, 1166
Bottorf, Jan, 416
Bounds-Checker, 386
boxing, 915
Bradberry, John L., 801
Brainmaker, 387
BREAK386, 220
Bright, Walter, 411
Brodie, Leo, 32
bStrings Library for C, 368
bugs, inheritable, 92
Building Block Software, 1179

C

"C Programming: A Thousand CURSES on TEXTSRCH," 269-273
"C Programming: CSORT: A Saga of a Sort," 369-373
"C Programming: Dear Al...,"959-963
"C Programming: DED Revisited and the Shaft," 1067-1071
"C Programming: Hacks, Spooks, and Data Encryption," 862-867
"C Programming: HYPERTREE: A Hypertext Index Technique," 560-564
"C Programming: Midi, Turbo C++, Token Pasting, and PC Hot Keys," 653-658
"C Programming: SD '90 and ANSI at Last,"464-469
"C Programming: TEXTSRCH connections," 175-178
"C Programming: TEXTSRCH Continued: The Expression Interpreter," 86-91
"C Programming: The B-tree Again," 1158-1163
"C Programming: The Past, the Future and Multi-mania," 765-770
"C++ File Objects," 506
"Circles and the Digital Differential Analyzer," 593-596
"Closing DOS's Backdoor," 906-909
"Compiler Supported DLLs for DOS," 407
"Controlling Background Processes Under Unix," 1097-1100

"CTrace A Message Logging Class," 1004-1008
C compiler, 868
C Programmer's Toolbox, 195
C programs, 312
C routine, 698
C Traps and Pitfalls, 91
C++ Compiler for 80386/486, 286
C++ debugging, 1000
C++ Primer, 477
C++ Programming Language, The, 477
C++ Rider, 1180
C++, 295, 477
    hot keys, 657
    release 2.0, 738
C++/Views, 1181
C, 243, 597, 678, 704
    binding to Lisp, 700
    code generator, 689
    extensions to, 709
    future of, 656
    library functions, 709
    memory, 682
    programs, 678
    teaching, 90
C-scape Interface Management Systems, 195
C/386 Optimizing Compilers and Tools Version 8.0, 887
C6 optimization, 918
C6, types of optimization, 920
C_talk/Views, 139
CAD/CAM Developer's Kit (CCDK), 1179
CalcDayOfWeek, 1072
California Scientific Software, 387
CALL, 0005, 906
calling functions, 406
Campise, Leon, 809
Canup, Bob, 125
Cartesian space, 880
casting, 1002
CD-ROM Developer's Lab, 578
CDP Communications, 368
Cedar Software, 1089
cells, 307
CGA animation, 21
chaos, 460
character shapes, 603
child, 471
circles, 42, 593, 892
circular functions, 976
CISC, 180
Class Four Transport, 1101
class, 233, 267, 1000
Classix, 983
clipping algorithm, 597
clipping, line segment, 597
CLOS, 265
CNS, 1181
Cobalt Blue, 157
Cobb Group, The, 185
code word decoding, 1113
    encoding, 1112
    formation, 1112
Codecheck, 285
CodeRunneR, 103
CodeTAP 386, 886
CodeView, 723
coding, superimposed, 994
Cohen-Sutherland, 597, 599
Coleman Softlabs, 886
color map, 122
Columbia University Center for Computing Activities, 388
Comm Toolbox, 1108
    sample application, 1109
Common Lisp, 83, 695
CommPort, 926
Communications Formulas and Algorithms, 1181
communications, 1108
Compiler Design in C, 480
compiler switch combinations, 922
compiler, 722, C, 868
    optimization, 917
    Pascal, 659, 868
compression, 1120
Computer Architecture: A Quantitative Approach, 969
Computer System Architects, 886
computer graphics, 42
Concept Dynamics, 982
Conger, Jim, 653, 961
Conley Computing, 285
Connection Manager, 1108
connectionism, 1063
constants, calculating, 979
content addressability, 295
control files, 1100
controllers, 607
cordic algorithms, 976
Coriolis Group, The, 285
COSMIC Software Catalog 1990 Edition, 388

CP/M, 907
Cramer, William D., 1004
Crocket, Kenneth L., 1101
Crockett, Dan W., 202, 292
CRYPTO program, 1070
CRYPTO, 863
CSORT, 369, API, 371
CTrace, 1006
CURSES, 269
CX Multiprocessing Extension Kit, 435
CyberArts International, 1089
Czuchry, Andrew J. Jr., 301

D

"DataFlow Multitasking," reference to, 18
"DDJ Hypertext Project, The," 489-491
"Debugging Memory Allocation Errors," 716-719
    reference to, 1095
"Demystifying 16-Bit VGA," 428-434
"Designing an OSI Test Bed," 1101-1107
"Drawing Character Shapes with Bézier Curves," 603-606
"Dynamic Link Libraries For DOS," 404
D-CC/88K, 287
DASM, 187
Data Encryption Standard, (DES), 864
    algorithm, 1067
Data Transforms Fontrix Format, 319
data acquisition, 26
data encryption, 38, 862
data segments, 227
data structures, 127. 417, 1001
data transfer, 1098
Datalight, 982
DataPak Software, 1180
DB-51 Enhanced System Monitor, 103
DDA, 593, 594
debugger, 364
debuggers, real-mode, 1021
debugging libraries, 1001
debugging, 220, 1004, C++, 1000,
    object-oriented, 1000
decryption, with DES, 865
deletion, 1120
DeMarco, Tom, 1170
dereferencing, null pointer, 1003
DES, 1067
    performance, 867
    permutations, 866
descriptor tables, protected-mode, 521
Desktop Stereo, 275
desktop manager, 1167
DESQview/X, 786
Developer Tools Express, 649
device driver, 1102
Devil's DP Dictionary, The, 580
Dewar, Robert B.K., 875
Dewhurst, Stephen C., 178
Diab Data, 287
DialogCoder, 104
dictionary, 994
Digital Research, 786
digital differential analyzer, 593
direct memory access, 26
DIVVY, 887
DLLs, 404
DMA, 26, interface circuit, building, 27
DOS extender, 895, 1019
DOS Protected Mode Interface (DPMI), 786
DOS, 376, 404, 613, 906
DosPTrace, 146
Douglas, Rohan, T. 36
DR DOS, version 5.0, 786
Drumlin, Inc., 887
Dudley, William F. Jr., 678
Duncan, Ray, 279, 969, 1170
Duntemann Jeff, 18, 92, 183, 202, 274, 293, 374, 394, 470, 565, 659, 771, 868, 964, 1072, 1164
Duvanenko, Victor, J., 597, 915
dynamic variables, 565

E

"Editorial," 583
"Editorial: A Little Help From Our Friends," 392
"Editorial: Bits and Bytes," 109
"Editorial: Conferences and Contests," 1093
"Editorial: Courting Trouble," 891
"Editorial: It Takes More Than Never," 291
"Editorial: Our 1991 Editorial Calendar," 675
"Editorial: Patent Letter Suits," 201
"Editorial: Putting HyperTheory Into Hyper Practice," 485
"Editorial: Running Light, And Still Without Overbyte," 17
"Editorial: School Days, Legal Maze," 987
"Editorial: Taking Care of Business, 791
"Encapsulating C Memory Allocation," 682-688
"Error Message Management," 36-37

"Examining Room: C_talk/Views," 139-144
"Examining Room: Cruising with TopSpeed," 333
"Examining Room: Encapsulating C++ Books," 477-478
"Examining Room: Examining Instant-C," 515-522
"Examining Room: Examining the Zinc Interface Library," 1122-1125
"Examining Room: Examining Zortech C++ 2.0," 43b-47
"Examining Room: Inside Object Professional," 818-822
"Examining Room: Inside Watcom C 7.0/386," 237-242
"Examining Room: Multiprocessing with Smalltalk/V," 435-440
"Examining Room: Optimizing with Microsoft C 6.0," 720-726
"Examining Room: Programmer Tools For Actor 3.0," 1026-1029
"Examining Room: The Power in PowerBasic," 619-623
"Examining Room: Unraveling Optimization in Microsoft C 6.0," 917-923
"Existential Dictionary, An" 994-999
"Extending printf()", 704-707
Earle, Daniel, 857
Eckel, Bruce, 727
ECOMP, 669
ED the Programmer's Editor, 1088
EdenSoft, 1088
edge enhancement system, 337
EES, 337
EGA adaptor, 23
Eiffel programming language, 805
Eikon Systems, 195
electronic data transmission errors, 1112
electronic mail, 1126, database system for, 1126
Electronics for Imaging, 669
Elements of Functional Programming, 388
ellipses, 595
Ellis, Graham, 709
embedded controllers, 79
embedded systems, 32
Empathy Inc., 983
EMS Toolkits, 787
encoding, BAM, 295,
    time, 376,
    date, 376
encryption, with DES, 864
Entsminger, Gary, 1122
Erickson, Jonathan, 17, 109, 201, 291, 392, 485, 583, 675, 791, 891, 987, 1093
Ericsson, Bo, 328
error correction, 1112
    detection, 1112
error message database, 36
errors, 135
Eurasip Workshop, 651
event managers, 1166
event-driven architecture, 1164
EXE file structure, 79
expanded memory, 413
ExploreNet 300, 983
expression interpreter, 86
Extender DialogHandler, 104
Extending DOS, 480
extensibility, 172

F

"Fast Search," 809-812
"Finite State Machines for XModem," reference to, 19
F77L-EM/32 Version 3.0, 887
FAR pointers, 523, 524
FASTRCH, 809
File Transfer Manager, 1109
file objects, 506
File_Spec, 506
FILESORT program, 371
Fischer, Ronald, 112
fixed table entries, 115
Flenniken, Steve, 113, 227
Floptical Drive, The, 479
Floyd, Edwin, 994
Floyd, Michael, 991
font data, noncompressed, 319
    maximum compression 319
fonts, designing, 604
Forth, 32
Fortran 77/386, 887
Fortran IV, 157
Fortran, 801
Fortran-77, 157
Foster, Ian, 287
Fractal Grafics, 1089
fractal geometry, 43b
fractal program, 463
fractals, 460, 677
Franz, Marty, 1026
free, 422
FreeForm, 887
Fried, Stephen, 441, 523
function communications, 135
function, 267,
    hash, 996,
    hyperbolic, 978

G

"Generation Scavenging," 396-403
"Generic One-Pass Assembler, A," 813-817
"Getting a Handle on Virtual Memory," 411-415
garbage collection algorithms, generation based, 399
Gaspar, Daon, 1108
GDT, 613
general protection, 48, 145
general registers, 443
generation based garbage collectors, 399
generation, 42
GEOCOMP, 1089
GEOGRAF Level One, 1089
Gessner, Rick, 492
Global Descriptor Table, (GDT), 613
global data, 405
Gold Hill Company, 368
Goldberg, Adele, 607
GoldWorks II, 368
GOSIP, 1101
Government Open Systems Interconnection Profile (GOSIP), 1101
GP faults, 48, 145
GPOStyle, 578
GrafPrint, 983
graphics, 511
Grasp, 670
GSpot, 983
GUI, 824
GUIDE, 104
Guide, release 3.0, 577
Gyurcsik, Ronald S., 597, 915

H

"Handling OS/2 Error Codes," 756-757
"Homegrown Debugging — 386 Style!," 220-226
handle pointers, 411, 413
Hardenbergh, Hal, 180
Harel, David, 460
hash function, 996
hashing algorithm, 176
HCR Corporation, 368
HCR/C++ Version 2.2, 368
header fields, 685
heap status, 718
heap, self adjusting, 128, 565
  packing, 568
  top-down-skew, 129
  pairing, 130
help engine, 504
help files, 503
help functions, for Lisp, 696
help system, 501
Help!, 1181
HELPMAKE, 502
Hennessy, John L., 969
heuristics, 127, algorithms, 127
Hewlett Packard, 121
hidden surface removal, 123
HighC Compiler Version 1.3, 103
HNC, 983
holes, 565
Howard, Christopher, 587
HP New Wave, 374
Hyde, Randall L., 232
hyperbolic functions, 978
HyperCard 2.0, 763
HyperCard, 555, 858
HyperEdit, 577
Hyperkinetix, 886
HyperLines, 557
HyperText problem, 763
hypertext engine, 498
hypertext systems, demystifying, 492
HyperText, 489, 492, 555, 560, 1155
  editing, 563
  editor, 494
  engine, 495
  in context, 501
  unit, 496
HyperTMON, 578
HyperTree, 560
  searching, 561
HyperWords, 557
HyperWriter, 489

I

"Implementing Bicubic Splines," 695-703
"Implementing Cordic Algorithms," 976-979
"Implementing the Rhealstone Real-Time Benchmark," 312-318
  reference to, 792
"Improving Line Segment Clipping," 597-602
"Information Models, Views, and Controllers," 607-612
I/O access speed, 434
IBM LinkWay: Hypermedia for the PC, 578
IBM, 328
ICOM Simulations, 578

idle function, 1110
Image Software, 670
in-line sort, 369
independent reinvention, 1016
Index Manager, 368
inheritance, 233, 966, 991
    multiple, 731
insertion, 1119
Inside Turbo C, 285
Inside Turbo Pascal, 285
Inside!, 918
Insite Peripherals, 479
Instant-C, 515
INT 21, 52, 906
    handler, 906
integer operation, 179
Intek C++ 2.0, 286
Intek C++, 393
Intel 286 protected Mode instructions, 518
Intel 80386 microprocessor, 237
Intel Corporation, 196
Intel's Personal Computer Enhancement Operation (PCEO), 787
interfaces, 487
interrupt handlers, in C, 225
interrupts, 523
Invention Software, 104
IRQ5, 52
ISAM, 1026
Ives, William E., 813

J

Jackson, Frank, 396
James, Rahner, 21
Jarvis, Pitts, 976
JBSoftware, 287
Jensen & Partners International, 333, 659, 868
JMP FAR, 906
John Wiley & Sons, 578
Johnson, J. Scott, 489
Johnston, J.C., 795
JPI, 386
JT Software, 982
JTW C++ class library, 982

K

K term, 965
Kar, Rabindra P., 312
KBM Communications, 368
Kelly-Bootle, Stan, 580
Kent Porter Scholarship Fund, 583
Kermit for OS/2, problems, 924
Kermit, 823
Kernighan, Brian, 689
King, Todd, 498, 603
Klimasauskas, Casimir C. "Casey," 337
Knowledge Systems, 1089
Knowles, Walter, 1030
Koenig, Andrew, 91
Kosko, Bart, 295

L

"Lessons from History: IBM Mainframe Error Handling," 150
"Letters," 292, 486, 18, 110, 393, 584, 676, 792, 892, 988, 1094
"Location is Everything!," 79-82,
    reference to, 292
"LZW Revisited," 553-554
Ladd, Scott Robert, 43b, 720, 805
Lahey Computer Systems, 887
LALR compiler, 386
LALR Research, 386
Lane, Alex, 333
language, object-oriented, 991
Lattice C 6.0, 270
Lattice CURSES, 270
Lauzzana, Raymond G., 695
Lawrence, Jeannette "Jet," 307
lawsuits, 1013
League for Programming Freedom, The, 1009
Letwin, Gordon, 145
Liao, Andrew M., 127
libraries, 404
    SEG4G, 616
    Turbo C, 727
    ZIL, 1122
Lindenmayer, Astrid, 1157
line segment clipping, 597
linking modules, 407
linking, 816
Lippman, Stanley B., 477
Lisp functions, high-level, 701
Lisp, 83
    what it is, 84
    what it isn't, 84
Lisp, 265

list, 266
Lister, Timothy, 1170
Liu, Hsi-Chiu, 1112
locate program, 80
Look & Feel Screen Designer, 195
LOOM systems, 416
Lyke, Daniel, 880
LZW compression program, 553

M

"Macintosh Communications Toolbox, The," 1108
"Making the Move to Modula-2," 795-800
"Managing Multiple Data Segments Under Microsoft Windows, Part I," 113-119
"Managing Multiple Data Segments Under Microsoft Windows, Part II," 227-231
"MCon Programming Interface Functions," 423
"Memory Controller, A," 422-427
"Multiplexing Error Codes," 135-138
"MVC Paradigm in Smalltalk/V, The," 1082-1086
Mac operating system, Version 7, 649, 650
MacApp, Version 2.0, 649
Macintosh Allegro Common Lisp L(CL) v. 1.3, 387
Macintosh-Aided Design magazine, 857
MacRenderMan, 669
MacWorld '90, 649
mainframe, 801,
    tape sort, 369
Mak, Nico, 756
malloc, 422
Margulis, Neal, 170
mark-and-sweep collection algorithms, 397
marketing software, 474
Master C, 787
mathematical manipulation, 976
MathPac1/V, 1089
maximum compression, of font data, 319
McGraw-Hill, 1181
MCI Mail, 1126
McMahon, William J., 135
Memory Commander, 982
memory, addressing, 870
    allocation, 990
    allocation errors, 716
    block header, 684
    display access speed, 432
    leakage, 682
    leaks, 1002
    models, 870
    systems, 295
    expanded, 413
    limitations, 411
    real, 445
    test-mode, 432
    VGA, 431
    virtual, 416
memory management, 422
    routines for, 422
menu generation, 1100
message logging, 1004
MetaWare, 103
Meyer, Bertrand, 805, 1075
Meyerowitz, Norman, 1156
Microprocessors: A Programmer's View, 875
Microsoft C 6.0, 917, 720,
    debugging, 723,
    pointers for optimization, 721
Microsoft C Professional Development System Version 6.0, 479
Microsoft C, version 6.0, 765
Microsoft Windows 2.0, 374
Microsoft Windows, 113, 195, 227, 1026
Microsoft, 758
Microsystems Software, 103
Microway, 1181
MIDI Sequencing in C and C Programming for MIDI, 653
Midi, 653
Mischel, Jim, 704
MMC AD Systems, 195
mode, protected, 441
    real, 441
Model-View-Controller (MVC), 607, 1082
Modula-2 Professional Version 2.1, 886
modula-2 books, 187
Modula-2, 183, 275, 333, 394, 565, 775, 795,
    libraries, 796
    team-programming, 797
modular multitasking, 187
Moeser, Robert A., 422
mouse control, 773
Move 'Em, 480
MS-DOS Kermit Version 3.0, 388
MS-DOS, 79
Multi-Threading Program Toolkit, 787
Multimedia Expo, 1181
Multiple Inheritance, 661
multiple data segments, 113

multiple inheritance, 731
multiprocessing, 435
multitasking, 765
multiuser, 765
MVC, 607,
reuse, 609
MVC, using, 142
myriad models, 871

N

"Neural Network Instantiation Environment, A," 301-306
NABIJAooc, 983
NABJA Software, 983
NDP C++, 1181
Nelson, Mark R., 79, 202
NEMO 2.0, 105
NetWare API, 1095
NetWare C interface, 767
NetWare environment, 767
NetWare Programmer's Workbench, 105
networking, 1098
neural nets, 307
neural nets, 1063
NeuralSource, The Bibliographic Guide to Artificial Neural Networks, 388
neurodes, 307
neuron model, 308
noise words, 176
Nolan, Tom, 26
noncompressed font data, 319
Notenboom, Leo, 501
Novell, 105
Nu-Mega, 386
null pointer dereferencing, 1003

O

"Object Swapping," 416-421
"Object-Oriented Debugging," 1000
"Object-Oriented Programming in Assembly Language," 232-238
"Of Interest," 103, 195, 285, 386, 479, 577, 669, 786, 886, 982, 1088, 1179
"Opening OS/2's Backdoor," 900-905
"Optimal Determination of Object Extents," 915-916
Oakland Group, 195
Object Pascal, 374
Object Professional, 818
object, extents, 915, hierarchy, 472
object oriented language, 991
object Prolog, 991
object swapping, processes involved, 418
Object-Oriented Design with Applications, 1075
Object-Oriented Software Constructions, 1075
object-oriented extension to C, 139
object-oriented numerical tools, 195
object-oriented programming, 92
object-oriented techniques, 470
objects, 805
objects, trouble with, 569
ObjectVision Inc., 787
ObjectVision Release 1.0, 787
Objectworks\C++ Release 2, 1088
Objectworks\Smalltalk Release 4, 1181
ODL parser, 993
Ohlsen, Chris, 1126
on the heap, 565
on-screen documentation, 501
one—pass assembler, 813
OOP, 111, 232, 470, 677, 1000,
debugging, 1000
OOPSLA, 89
optimization, aliasing in, 920
Optronics of Ashland, 275
Orchid, 480
OS/2, 146, 823, 900, 924
device chain, 901
error codes, 756
limitations, 924
mini-primer, 823
OS/2 Presentation Manager, 195
OS2ERR, 756,
using, 756
OSI reference model, 1105
OSI, 1101
Overlay Architect, 103
Overlay Designer, 886
Overlay Optimizer, 103
OWL International, 577

P

"Parallel Extensions to C," 709-715
"Parametric Circles," 42-43
"Parker's Perceptions," reference to, 19
"Persistent Objects in Turbo Pascal," 805-808
"Pick-A-Number Interfaces," 125-126
"Pixel Ordering Algorithm, A," 511-514
"Porting C Programs to 80386 Protected Mode," 678-681

"Porting Fortran Programs from Minis to PCs," 801-804
reference to, 1094
"Programmer Paradigms: The Promise of System 7," 955-958
"Programmer's Bookshelf: Microprocessors From the Programmer's Perspective," 875-876
"Programmer's Bookshelf: Object-Oriented Software Development: Reality Sets In," 1075-1077
"Programmer's Bookshelf: Taking Card of Business," 1170-1171
"Programmer's Bookshelf: The Theory and Practice of Computer Design and Implementation," 969-970
"Programmer's Workbench: A Database System for Automating E-Mail," 1126-1131
"Programmer's Workbench: Accessing Hardware from 80386 Protected Mode Part II," 523-527
"Programmer's Workbench: Accessing Hardware from 80386 Protected Mode, Part I," 441-446
"Programmer's Workbench: Collections in Turbo C++," 727-732
"Programmer's Workbench: DOS + 386 = 4 Gigabytes," 613-618
"Programmer's Workbench: Kermit for OS/2 Part I," 823-826
"Programmer's Workbench: Kermit for OS/2 Part II," 924-928
"Programmer's Workbench: Mixed-Language Programming with ASM," 243-247
"Programmer's Workbench: Neural Networks and Image Processing," 337-342
"Programmer's Workbench: Stalking GP Faults: Part I," 48-54
"Programmer's Workbench: Stalking GP Faults: Part II," 145-151
"Programmer's Workbench: Windows 3.0 Application Development," 1030
"Programming Paradigms: A Taste of Lisp," 83-85
"Programming Paradigms: Complex Systems, Fractals, and Chaos," 460-463
"Programming Paradigms: Cooperation and Competition," 649-652
"Programming Paradigms: Do You Have a Future in Vegative Virtuality?," 857-861
"Programming Paradigms: Four Hundred and Eighty-seven," 172-174
"Programming Paradigms: Getting CLOS," 265-268
"Programming Paradigms: HyperCard and (or?) Hypertext," 555-559
"Programming Paradigms: Neural Nets: A Cautionary View," 1063-1066
"Programming Paradigms: Windows 3.0 Challenges All the Talent in the Room," 758-764
"Programming Paradigms: Wrapping Up Loose Ends," 1154-1157
"Programming Paradigms: Wrapping Up Software Development '90," 364-367
"Programming RISC Engines," 170
"Protected-Mode Operations on a PC," 898
PAL, 1126,
script, 1130
Paradigm Systems, 918
Paradox Application Language, 1126
Paradox engine application, 1126
Paradox Engine, 285
PARAGen, 982
Parallel Distributed Processing, 1063
parallelpiped, 915
ParcPlace Systems, 1088, 1181
parent, 93, 471, 677
partial polygons,122
Pascal 5.5, 653
Pascal compiler, 868
Pascal, 243, 565, 795,
compilers, 659, 868
patent system, 1009
Patents, 584
patents, absurd, 1009
obvious, 1011
searches, 1013, 1014
software, 486
obscure, 1015
Paterson, Tim, 227, 593
paths, 1164
Patterson, David A., 969
Patterson, Tim, 113
Paul Mace Software, 670
PC DMA Architecture, 26
PC Hot Keys, 653
PC Techniques, 285
PC-Browse, 577
PC-lint Version 4, 887
PDC Prolog, 991
PDP, 1063
Peitgen, Heinz-Otto, 461
Penrose, Denise E.M., 695
Peopleware: Productive Projects and Teams, 1170
Peritus International, 286
permutations, DES, 866

Phase One, 1088
Phoenix Technologies, 196
Pixar, 669
Pixelab, 983
Pocket Soft, 195
pointer alignment, 683
pointers, 892
polymorphic compilation, 868
polymorphism, 95, 235
port, 900
Portable Fortran Code, writing, 803
Porter Kent, 583
PowerBasic, 619
    language enhancements, 620
    benchmarks, 620
PowerCell Spreadsheet Library, 786
Prentice Hall, 480
printf(), 704
privacy, 967
processing elements, 307
ProDesigner II VGA, 480
programing, challenge, 365
Programming in C++, 178
Prograph 2.0, 1154
Prolog Development Center, 285
Prolog, 649
PROT, 895, 1019
Protected Virtual Address Mode, 48
protected mode, 441, 678
    descriptor tables, 521
    interpreter, 516
    peek/poke, 49
Prusinkiewicz, Przemyslaw, 1157
PS/2, 328

Q

Qualitas, 274, 480
Quantasm Corporation, 286
Quantasm Power Lib, 286
Quarterdeck, 786
queues, 769
QuickMod, 886
QuickPascal, 274, 374
Quicksort, 577

R

"RAM Disk Driver for Unix." 901-914
"Ray Tracing," 880-883
    reference to, 988
"Real-Time Animation," 21-26
    reference to, 292
"Real-Time Data Acquisition Using DMA," 26-31
"Roll Your Own DOS Extender: Part I," 895-899
"Roll Your Own Object-Oriented Language," 991-993
"Roll Your Own: DOS Extender: Part II," 1019-1025
Raima, 786
RAM disk driver, 910
RAM disk, adding to Unix, 912
RAM vs. ROM, 34
ration Technologies, 286
Rational Systems, 515
read-only data, 227
Reade, Chris, 000
Ready Systems, 285
Reagen, Jeff, 910
real memory, 445
real mode, 50, 441
    debuggers, 1021
    graphics driver, 680
recursion, 172
Regan, Shawn M., 553
register-aliased variables, 525
registers, general, 443
regular expressions, 690
replacement functions, 716
Resource Workshop, 1088
Rhealstone, 488
    components, 313
    real-time benchmark, 312
RISC, 170, 876
    branch and call instructions, 181
Rix Softworks, 669
RIXAI, 669
Robbins, W.E., 597, 915
Rogue Wave, 195
ROM, 431
ROM-DOS Developer's Kit, 982
Rorabaugh, C. Britton, 1181
Rothstein, Arthur, 901
RTUX, 801

S

"S-CODER for Data Encryption," 38-41
"Self-Adjusting Data Structures," 127-134
"Self-Referential Hypertext Engine, A," 498-500
"Software Patents," 1009-1018

"Standardizing the Standardizing Process," reference to, 293
"Structured Programming: Chasing Bubbles in the Waterbed," 565-570
"Structured Programming: Dressing the Truth in Rubber Suits," 92-96
"Structured Programming: Grinding the Speckled Axe," 470-474
"Structured Programming: Ice Cubes in the Swimming Pool," 1072
"Structured Programming: Pieces of Charlie," 868-874
"Structured Programming: Sex and Algorithms," 964-968
"Structured Programming: Sifting for Sharks' Teeth," 274
"Structured Programming: The Cuba Lake Effect," 1164
"Structured Programming: The Day the Earth Wouldn't Sit Still," 183-188
"Structured Programming: The Just One Thing Dilemma," 659-664
"Structured Programming: Yes Virginia, There Is a Xerox!," 374-380
"Structured Programming: You See Toothpaste Pumps; I See Coil Forms. . .," 771-775
"Super VGA Programming," 587-592
"Supercharging Sequential Searches," 1116-1121
"Swaine's Flames: Caution! Man at Work," 671
"Swaine's Flames: A Look at the Latest Videos," 1182
"Swaine's Flames: How To Succeed In the Window Business Without Really Trying," 1090
"Swaine's Flames: Hyperterminology from Hell," 580
"Swaine's Flames: Is Copyrighting Software Futile?," 984
"Swaine's Flames: Junk Customers," 389
"Swaine's Flames: Pub Crawler," 288
"Swaine's Flames: Standing in the Rubble, Looking Up," 106
"Swaine's Flames: The Integrity of the Product," 788
"Swaine's Flames: The War on Bugs," 198
"Swaine's Flames: Thoughts on the Reviewing of Software," 868
"Swaine's Flames: Three on a Match," 481
S&H Computer Systems, 982
S-Coder algorithm, 486
S-CODER, engine, 39,
applications, 40
S20 Developpement, 105
Sageline, 578
Sapiens Software, 195
Satchell, Stephen, 216
SATVU, 1088
Saupe, Dietmar, 461
Schatzman, Bruce D., 917, 721
Schell, Rick, 243
Schidlt, Herb, 653
Schimandle, Jim, 682
Schroff Development Corporation, 983
Schulman, Andrew, 48, 145, 237, 407, 477, 875, 1075
Science of Fractal Images, 461
Scientific American, 652
screen module, 926
searching, 1117
Searfass, Glenn, 319
SEG4G library, 616
segment table, 115
SEGTABLE, 794
sequential search algorithm, 1116
sequential searches, 1116
shell implementation, 685
Sierra C toolset for the M68000, 285
Sierra Systems, 285
Silco-Magnetic Intelligence, 480
SilverComm C Async Library, 1179
Silverscreen, 983
SilverWare, 1179
slow curve, 989
Smalltalk-80, 610
Smalltalk/V 286, 435, 1082
vs. Smalltalk-80, 1082
smart line editor, of ZIL, 1124
Smosna, Matthew, 875
Soft As It Gets, 1088
Softaid, 394
Software Development '90, 464
Software Development Systems, 887
Software Mart, 578
Software Organization, The, 104
Software Translations Inc. (STI), 104
software, marketing, 474
modularity, 303
software patents, 486
difference, 1012
Solid Link Library, 1181
Solid Software, 1181
sort program, 369
sort routine, 619
sound generator, 526
spectrum-analyzer, 243
SpeedyWrite Software, 577
Spencer, Lawrence D., 716
Spinnaker PLUS Software slot Developer's Kit for the Macintosh, 1088

Spinnaker Software, 1088
splay trees, 132
Splines, 695
stack frame, 243
stamps, 376
standard library, 683
Star Sapphire Common Lisp Version 3.0, 195
Stark, Kathy T., 178
Starting Forth, 32
Stevens, Al, 18, 20, 86, 110, 175, 203, 269, 369, 464, 560, 765, 862, 1067, 1158
Stony Brook Modula-2, 183
Stony Brook Professional Modula-2 Version 2.0, 388
Stony Brook Software, 388, 659, 886, 924
Stout, Robert B., 38
Strand: New Concepts in Parallel Programming, 287
Stroustrup, Bjarne, 477, 709
Stroyan, Michael, 120
structural self-adjusting heuristic, 127
Sun Microsystems, 104
Super VGA BIOS Extensions, 196
Super VGA, 988
superimposed coding, 994
SuperScript, 558
Swaine, Michael, 20, 83, 106, 172, 197, 198, 265, 288, 364, 389, 460, 481, 555, 649, 671, 758, 788, 857, 868, 955, 984, 1063, 1090, 1154, 1182
swapping, objects, 416
Switzer, John, 906
Syck, Gary, 404
Symantec, 479
Symbols, 113
symbols, 173
symbols, 813
    management, 815
System 7, 955
    vs. System 6, 759
System Apps, building, 958
systematic error handling, 135

T

"The Missing Link: Why We're All Doing Hypertext Wrong," 1156
"Three-Dimensional Graphics Using the X Window System," 120-124
Taylor, Stephen, 287
Teach Yourself C++, 653
TEMPFILE.DAT, 506
Terminal Manager, 1108
test bed, 1101
TEXTSRCH, 86, 175, 269
    performance, 273
    running, 177
TGSSystem, 1154
The Algorithmic Beauty of Plants, 1157
The Builder Version 1.21, 886
The Waite Groupe Press, 787
Think Pascal Version 3.0, 479
Thompson, Michael T., 1101
three dimensional, 880, 915
TIFF Development Library for PCs, Version 5, 670
Token Pasting, 653
Tonkin, Bruce, 619
Tooke, Simon, 1000
ToolBook vs. HyperCard, 762
Toolbook Author's Resource Kit, 1179
ToolBook, 1030, 1031
tools, 722
Top Level, 887
Top Speed Modula-2 Version 2.00, 386
TopCL Version 2.0887
topology, 308
TopSpeed C Version 1.02 Extended Edition, 333
    compiling, 334
    project files, 335
    TechKit, 335
TopSpeed C, 372, 677, 963
TopSpeed Modula-2 objects, 966
Tosh Systems, 787
Tracy, Martin, 32
training algorithm, 308
    supervised, 310
    unsupervised, 310
transforming 3-D to 2-D, 121
transputer architecture, 711
transputers, 709
Tronix, 103
TSR Systems, 1088
TSR, 820
TSX-32 Version 2.1, 982
Turbo C++, 653
Turbo C++, 669
Turbo C++, 727
Turbo C++, 765
Turbo C++, 893
Turbo Debugger and Tools, 464
Turbo Debugger Version 2.0, 386
Turbo Pascal 5.5, 374

Turbo Pascal 6.0, 1166
Turbo Pascal, 4.0, 1073
Turbo Pascal, 805
Turbo Power Software, 818
Turbo Professional, 730, 818
Turbo Prolog, 285
Turbo Vision, 1166

U

"Unbundled Integration," reference to, 19
"Undocumented DOS," reference to, 111
"Untangling Neural Nets," 307-311
UniFLEX Computing, 1179
UniFlEX.wks, 1179
units, 307
University of Georgia, 388
Unix System V/386 Release 3.2, 910
Unix, 910, 1097
Unix/Xenix Kernel Debugger (KDB), 103

V

"VESA VGA BIOS Extensions," 328
V Communications, 982
Van Norstrand Reinhold, 388
variable arguments, in C, 704
variable *pSegs*, 116
VDSA BIOS extension, 330
Version 7, Mac operating system, 649, 650
VGA, 328, 587, 16-bit, 428
    memory, 431
    performance, 428
Video Electronics Standards Association (VESA), 196
video graphics array (VGA), 328
views, 607
virtual memory, 411, 416
virtual methods, 966
VMS, 801
Vose, Michael, 501
VRTX-PC, 285

W

wait states, 430
WAN, 1101
Wasserman, Phillip, 388
Watcom and Novell, 239
Watcom C 7.0/386, 237
Watcom compiler, 678
Watcom Optimizing Compiler and Tools Version 8.0, 577
Watcom, 333, 577, 765, 887
Weeks, Kevin, 506
Weinberger, Peter, 689
Western Wares, 1180
Whitewater Group, 1026, 386
Whitewater Resource Toolkit, 1026, 1027
Wide Area Network, 1101
Widgets/V 286, 1089
William Horton Associates, 577
Williams, Al, 220, 613, 1019, 1116
window, 120
    framing, 1085
    constructing, 1085
windowing, 597
Windows 3.0, 758, 794, 1030, vs. DOS, 758
    vs. Windows 2.0, 759
    vs. System 7, 761
WinTrieve, 1026, 1027
Wirth, Nick, 186
Word Solution Engine, 1180
Wright, Karl, 243
Write*On, 578

X

X window system, 120
X3J13, 265
Xerox Palo Alto Research Center, 416
Xerox PARC Ooze, 416

Z

"ZEN for Embedded Systems," 32-35
"Zen Forth," reference to, 394
Zale, Bob, 619
Zaron Software, 1181
ZEN, running, 35
    word set, 33
Zigon, Robert, 42, 203, 593
ZIL, 1122
Zinc Interface Library, 787, 1122
Zince Software, 787
Zortech C++ 2.0, 43b, 467, 765
    integrated environment, 44
    AT&T compatibility, 45
    documentation, 46
Zortech C++ Development System for MS-DOS and
    OS/2, Version 2.1, 669
Zortech, 333

The Complete *Dr. Dobb's Journal* Bound Volumes

Each bound volume contains a full year's worth of useful code and fascinating history from *Dr. Dobb's Journal*. *DDJ* is the oldest and most popular programmer's magazine today, and many back issues have long been sold out. Most of the practical technical information contained in these volumes is not available from any other source. But within these giant volumes, you'll find a treasury of useful source code and programming tips.



Volume 1 — 1976
Chronicles the advent of the microcomputer era in 1976. Always pertinent for bit crunching and byte saving, home-brew computer projects, and the technical history of home computing. Topics include: Tiny BASIC, the first words on CP/M, speech synthesis, floating point routines, the 6502 disassembler for the Apple, and much more. 364 pp.

Volume 2 — 1977
The small computer emerges as a powerful tool for the modern age in these issues from 1977. Topics include: Lawrence Livermore Lab's BASIC, Dr. Starweather's PILOT, using a modem, string-handling techniques, ciphers, Turtle graphics, and micro utilities. 498 pp.

Volume 3 — 1978
Explores the foundation of the mass-market computer industry in 1978. Topics include: programming in BASIC, PILOT, and Pascal; RAM memory testers; Apple utility programs; the S-100 bus standard; STRUBAL; and a structured BASIC compiler. 478 pp.

Volume 4 — 1979
Heralds the widespread acceptance of the microcomputer in America. Innovative ideas and articles guide the reader through the age of discovery in 1979. Topics include: selecting business software, microcomputer speech and music, information networks, and interfacing techniques. 467 pp.

Volume 5 — 1980
Focuses on the technological promise of the modern microcomputer. Topics include: the revolutionary impact of CP/M, C programming and the UNIX operating systems, a survey of computer networks, software portability, introduction to Forth, and compiler writing. 450 pp.

Volume 6 — 1981
The microcomputer enters the mainstream. These issues assess the revolutionary impact of microcomputers on the individual and on society in 1981. Topics include: computer conferencing, the power of Forth, B- and 16-bit technology, Rubik's cube simulator, and an adventure game development systems. 558 pp.

Volume 7 — 1982
Examines the potential of powerful 16-bit micros, including the birth of the IBM PC in 1982. Topics include: in-depth coverage of Forth, 68000, 8088 programming, C software tools, utilities for CP/M—including a spelling checker, using bulletin boards, and more. 568 pp.

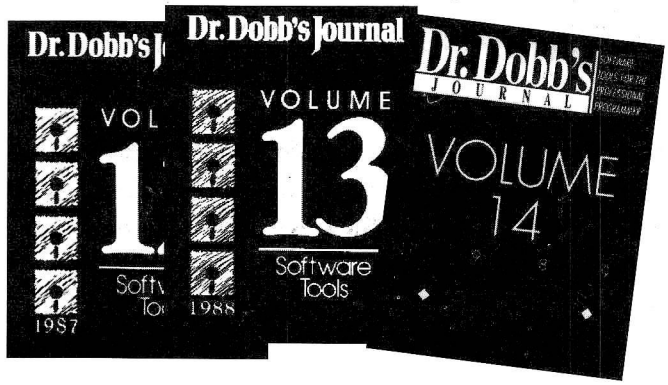
Volume 8 — 1983
DDJ turns pro. Some of the most powerful professional programmer's tools ever published in a magazine are in this 1983 volume, including Small-C, the RED editor, and an Ada subset. 798 pp.

Volume 9 — 1984
Shaping things to come. In 1984 *DDJ* examined new programming environments including Prolog, expert systems, Modula-2, and Pascal. Other topics include GREP, UNIX internals, and two encryptions systems. 982 pp.

1-800-533-4372 (in CA 1-800-356-2002)

M&T BOOKS

More Bound Volumes



Volume 10 — 1985

The year of living dangerously. In 1985 *DDJ* added more memory, an SCSI port, and a hard disk to the Macintosh; challenged the UNIX establishment with plans for a free UNIX. Includes software tools in C, Modula-2, Forth, Pascal, assembly language, and Prolog. 942 pp.

Volume 11 — 1986

The promise of power. Desktop computers began to rival minis and mainframes in power. *DDJ* covered 1986's changes with issues on the 68000, parallel processing, artificial intelligence, the 80386, and multitasking. *DDJ* also supported the new chips with assemblers, translators, and other development tools. 868 pp.

Volume 12 — 1987

Building better brains. *DDJ* broke new ground in 1987 with a neural network implementation, monthly coverage of artificial intelligence and object-oriented programming techniques, and reviews of implementation of Prolog, LISP, and Smalltalk. 1015 pp.

Volume 13 — 1988

The new frontier. Database technology, Hyperanimation, and much more! *DDJ* talks about how to create a new language, writing real time programs under UNIX, and developing good software. Other topics on object-oriented programming, SQL development tools, and much more are all included. 864 pp.

Volume 14 — 1989

Wrapping up the 1980's. Highlights include complete coverage of object-oriented programming, windowing systems, 32-bit programming, memory management, graphics programming, and a host of other exciting topics. *DDJ* also delivered two special issues — *Dr. Dobb's Macintosh Journal* and *Dr. Dobb's C Sourcebook*, both are included in this volume. 1051 pp.

To Order: Return this order form with your payment to: M&T Books, 501 Galveston Drive, Redwood City, CA 94063 or **CALL TOLL FREE 1-800-533-4372 (in California, CALL 1-800-356-2002).**
FREE POSTAGE: Order three or more bound volumes and the postage is FREE!

❑ **YES!** Please send me the following:

❑ Vol. 1	$15.50	❑ Vol. 9	$15.50
❑ Vol. 2	$15.50	❑ Vol. 10	$15.50
❑ Vol. 3	$15.50	❑ Vol. 11	$15.50
❑ Vol. 4	$15.50	❑ Vol. 12	$19.87
❑ Vol. 5	$14.50	❑ Vol. 13	$19.88
❑ Vol. 6	$14.50	❑ Vol. 14	$19.89
❑ Vol. 7	$14.50	❑ Vol. 1 – 14	$192.25
❑ Vol. 8	$14.50		

Subtotal _____
CA residents add applicable sales tax ___% _____
Shipping: Free on domestic orders of three _____
or more volumes. See shipping rates. _____
TOTAL _____

Name _____
Address _____
City _____
State _____ Zip _____
❑ Check enclosed, payable to M&T Books.
Charge my ❑ Visa ❑ M/C ❑ AmEx
Card No. _____
Exp. Date _____
Signature _____

7069

Shipping Rates
Domestic: Add $6.00 per volume. Postage on orders of 3 volumes is FREE!
Foreign: Add $12.00 per volume surface mail.

1-800-533-4372 (in CA 1-800-356-2002)

Just the Fax...

SAVE 40%!

Start your *Dr. Dobb's Journal* subscription today by faxing this order form. It will be processed the day we receive it. That means you'll benefit from the industry's top source for professional programmers that much sooner. No ifs, ands, or buts. Just the fax.

Dr. Dobb's JOURNAL

SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

TO:

Name: ***Dr. Dobb's Journal Subscription Department***

Fax number: ***(415) 364-8630***

FROM: ***(please print)***

Name: _____

Address: _____

City/State/Zip: _____

YES! Enter my one-year subscription to *Dr. Dobb's Journal* (12 information-packed issues) and bill me later for only $25. I save 40% off the cover price.

For your convenience, you may charge your subscription on your

☐ VISA or ☐ MasterCard

Card Number: _____ Exp. Date: _____

Signature: _____

Savings based on the full-year newstand rate of $42. Basic annual subscription rate: $29.97. Foreign subscriptions must be prepaid in U.S. dollars drawn on a U.S. bank or by VISA or MasterCard. Canada and Mexico: $45/year (surface mail). All other foreign countries: $70/year (air mail). California residents please add 8.25% sales tax. Canadian GST included (#R124771239). Please allow 4-6 weeks for delivery of first issue.

Offer expires December 31, 1992

5DJB

Dr. Dobb's Journal, 501 Galveston Drive, Redwood City, CA 94063

Just the Fax...

SAVE 30%

Start your *Microsoft Systems Journal* subscription today by faxing this order form. It will be processed the day we receive it. That means you'll benefit from the magazine designed to enhance your software development skills that much sooner. No ifs, ands, or buts. Just the fax.

MICROSOFT SYSTEMS JOURNAL

MSJ

TO:

Name: **MICROSOFT SYSTEMS JOURNAL SUBSCRIPTION DEPARTMENT**

Fax number: **(415) 364-8630**

FROM: ***(please print)***

Name: _____

Address: _____

City/State/Zip: _____

YES! Enter my one-year subscription to *Microsoft Systems Journal* (6 issues) and bill me later for only $34.95. I save 30% off the basic subscription price.

For your convenience, you may charge your subscription on your

☐ VISA or ☐ MasterCard

Card Number: _____ Exp. Date: _____

Signature: _____

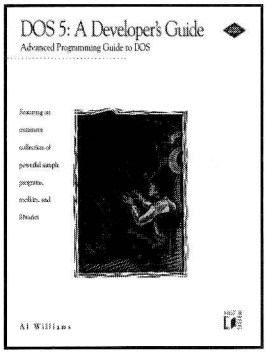
Basic annual subscription rate: $50. Foreign subscriptions must be prepaid in U.S. dollars drawn on a U.S. bank or by VISA or MasterCard. Canada and Mexico: $65/year (surface mail). All other foreign countries: $70/year (air mail). California residents please add 8.25% sales tax. Canadian GST included (#R124771239). Please allow 4-6 weeks for delivery of first issue. Guarantee: If you are ever dissatisfied with Microsoft Systems Journal, you're entitled to a full refund on all unmailed issues.

5MSB

Offer expires December 31, 1992

Microsoft Systems Journal, 501 Galveston Drive, Redwood City, CA 94063

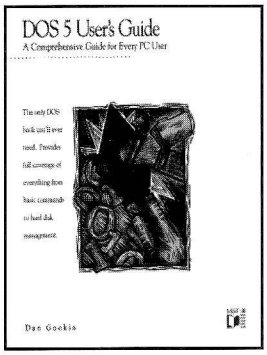
A Library of Technical References from M&T Books



DOS 5: A Developer's Guide
Advanced Programming Guide to DOS
by Al Williams

This complete guide to programming powerful, professional applications for all versions of MS-DOS covers such advanced topics as direct access techniques, fundamentals of graphics, expanded and extended memory, TSR programming, protected mode, and DOS extenders. Packed with solid examples programmers can use, study, and modify, it even includes the complete source code for an 80386 DOS extender. All of the source code is available on disk in MS/PC-DOS format. 914 pp.

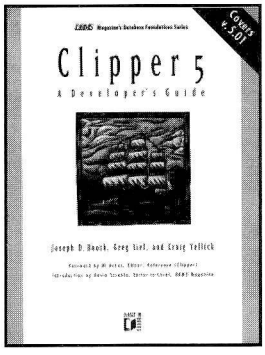
Book/Disk (MS-DOS)	**Item #179-2**	**$39.95**
Book only	**Item #177-6**	**$29.95**



DOS 5 User's Guide: A Comprehensive Guide for Every PC User
by Dan Gookin

Take control of the MS-DOS® operating system with this complete guide to using the world's most popular operating system. *DOS 5 User's Guide* contains clear, concise explanations of every feature, function, and command of DOS 5.0. Novice PC users will gain a quick start on using DOS, while advanced users will learn savvy tricks and techniques to maneuver their way quickly and easily through the system. Practical discussions and helpful examples teach readers how to edit text files, use directories, create batch files, and much more. Advanced topics include using EDLIN, the DOS text editor; configuring the system; and using the DOS shell. 771 pp.

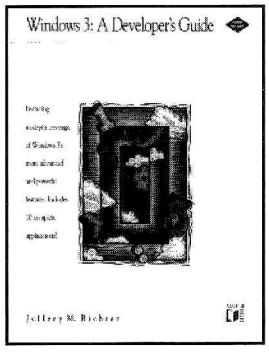
Book only	**Item #188-1**	**$24.95**



Clipper 5: A Developer's Guide
by Joseph D. Booth, Greg Lief, and Craig Yellick

An invaluable guide for all database programmers developing applications for Clipper® 5. Provides a quick introduction to Clipper 5 basics and discusses common programming needs such as designing data files, user interfaces, reports, and more. Advanced topics include networking, debugging, and pop-up programming. Code examples are used throughout the text, providing useful functions that can be applied immediately. All source code is available on disk in MS/PC-DOS format. 1300 pp. approx.

Book & Disk (MS-DOS)	**Item #242-X**	**$44.95**
Book only	**Item # 240-3**	**$34.95**



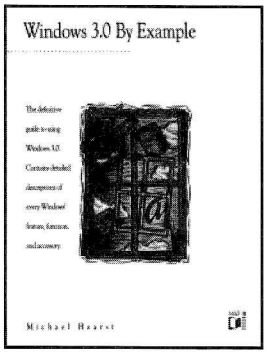
Windows 3: A Developer's Guide
by Jeffrey M. Richter

This example-packed guide is for all experienced C programmers developing applications for Windows 3.0. This book describes every feature, function, and component of the Windows Application Programming Interface, teaching programmers how to take full advantage of its many capabilities. Diagrams and source code examples are used to demonstrate advanced topics, including window subclassing, dynamic memory management, and software installation techniques. 671 pp.

Book/Disk (MS-DOS)	**Item #164-4**	**$39.95**
Book only	**Item #162-8**	**$29.95**

1-800-533-4372 (in CA 1-800-356-2002)

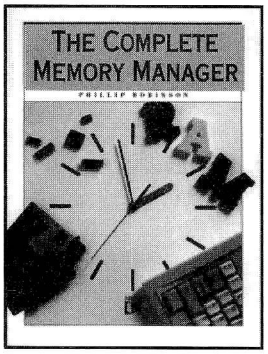
A Library of Technical References from M&T Books



Windows 3.0 By Example
by Michael Hearst

Here is a hands-on guide to Windows 3.0. Written for all users new to Windows, this book provides thorough, easy-to-follow explanations of every Windows 3.0 feature and function. Numerous exercises and helpful practice sessions help readers further develop their understanding of Windows 3.0. 398 pp.

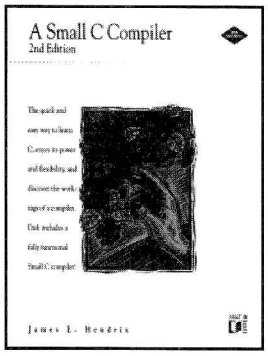
Book only **Item #180-6** **$26.95**



The Complete Memory Manager
Every PC User's Guide to Faster, More Efficient Computing
by Phillip Robinson

Readers will learn why memory is important, how and when to install more, and how to wring the most out of their memory. Clear, concise instructions teach users how to manage their computer's memory to multiply its speed and ability to run programs simultaneously. Tips and techniques also show users how to conserve memory when working with popular software programs. 437 pp.

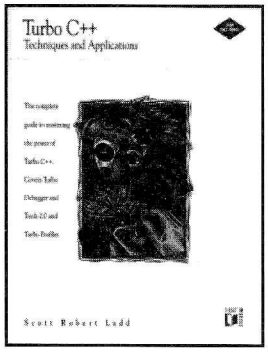
Book only **Item #102-4** **$24.95**



A Small C Compiler, Second Edition
by James Hendrix

This is a solid resource for all programmers who want to learn to program in C. It thoroughly explains Small C's structure, syntax, and features. It succinctly covers the theory of compiler operation and design, discussing Small C's compatibility with C, explaining how to modify the compiler to generate new versions of itself, and more. A fully-working Small C compiler, plus all the source code and files are provided on disk in MS/PC-DOS format. 628 pp.

Book/Disk (MS-DOS) **Item #124-5** **$34.95**



Turbo C++ Techniques and Applications
by Scott Robert Ladd

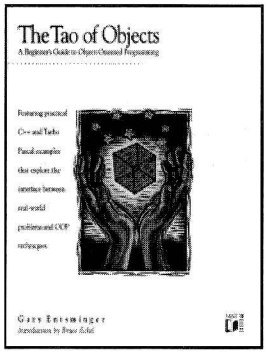
Turbo C++ Techniques and Applications is a complete guide and tutorial to mastering Turbo C++™, an object-oriented extension of the popular C language. Written for proficient C programmers, this book takes readers from the fundamentals of Turbo C++ to the inner workings of the language, thoroughly explaining its many features. Programmers will discover the how, what, and why of Turbo C++ programming and gain hands-on experience through a series of entertaining programming examples. Comprehensive discussions and numerous tips, tricks, and techniques detail Turbo C++'s commands and functions, teaching programmers how to write efficient code that fully utilizes the speed and energy of Turbo C++. All source code is available on disk in MS/PC-DOS format. 423 pp.

Book/Disk **Item #146-6** **$39.95**

Book only **Item #147-4** **$29.95**

1-800-533-4372 (in CA 1-800-356-2002)

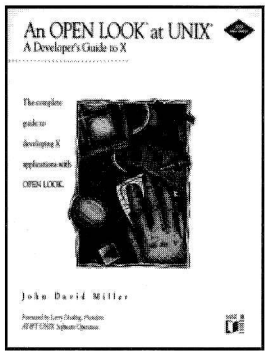
A Library of Technical References from M&T Books



The Tao of Objects: A Beginner's Guide to Object-Oriented Programming by Gary Entsminger

The Tao of Objects is a clearly written, user-friendly guide to object-oriented programming (OOP). Easy-to-understand discussions detail OOP techniques teaching programmers who are new to OOP where and how to use them. Useful programming examples in C++ and Turbo Pascal illustrate the concepts discussed in real-life applications. 249 pp.

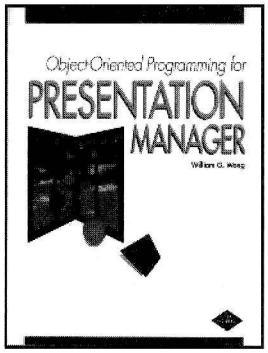
Book only	**Item #155-5**	**$26.95**



An OPEN LOOK at UNIX: A Developer's Guide to X by John David Miller

This is the book that explores the look and feel of the OPEN LOOK graphical interface, discussing its basic philosophy, environment, and user-interface elements. It includes a detailed summary of the X Window System, introduces readers to object-oriented programming, and shows how to develop commercial-grade X applications. Dozens of OPEN LOOK program examples are presented, along with nearly 13,000 lines of C code. All source code is available on disk in 1.2 MB UNIX cpio format. 482 pp.

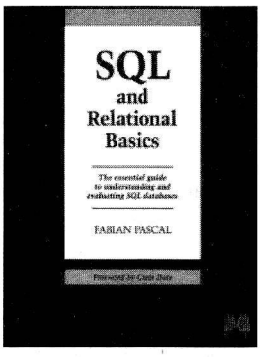
Book/Disk (UNIX cpio)	**Item #058-3**	**$39.95**
Book only	**Item #057-5**	**$29.95**



Object-Oriented Programming for Presentation Manager by William G. Wong

Written for programmers and developers interested in OS/2 Presentation Manager (PM), as well as DOS programmers who are just beginning to explore Object-Oriented Programming and PM. Topics include a thorough overview of Presentation Manager and Object-Oriented Programming, Object-Oriented Programming languages and techniques, developing Presentation Manager applications using C and OOP techniques, and more. 423 pp.

Book/Disk (MS-DOS)	**Item #079-6**	**$39.95**
Book only	**Item #074-5**	**$29.95**



SQL and Relational Basics by Fabian Pascal

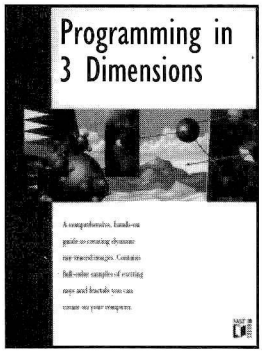
SQL and Relational Basics was written to help PC users apply sound and general objectives to evaluating, selecting, and using database management systems. Misconceptions about relational data management and SQL are addressed and corrected. The book concentrates on the practical objectives of the relational approach as they pertain to the micro environment. Users will be able to design and correctly implement relational databases and applications, and work around product deficiencies to minimize future maintenance. 336 pp.

Book only	**Item #063-X**	**$28.95**

1-800-533-4372 (in CA 1-800-356-2002)

M&T BOOKS

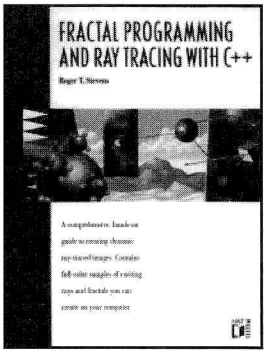
A Library of Technical References from M&T Books



Programming in 3 Dimensions: 3-D Graphics, Ray Tracing, and Animation by Sandra Bloomberg

Programming in 3 Dimensions is a comprehensive, hands-on guide to computer graphics. It contains a detailed look at 3-D graphics plus discussions of popular ray tracing methods and computer animation. Readers will find techniques for creating 3-D graphics and breath-taking ray-traced images as well as explanations of how animation works and ways computers help produce it more effectively. Packed with examples and C source code, this book is a must for all computer graphics enthusiasts! All source code is available on disk in MS/PC-DOS format. Includes 16 pages of full-color graphics. 500 pp. approx.

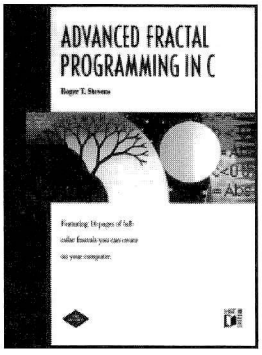
Book /Disk (MS-DOS)	**Item #218-7**	**$39.95**
Book only	**Item #220-9**	**$29.95**



Fractal Programming and Ray Tracing with C++ by Roger T. Stevens

Finally, a book for C and C++ programmers who want to create complex and intriguing graphic designs. By the author of three best-selling graphics books, this new title thoroughly explains ray tracing, discussing how rays are traced, how objects are used to create ray-traced images, and how to create ray tracing programs. A complete ray tracing program, along with all of the source code, is included on disk in MS/PC-POS format. Contains 16 pages of full-color graphics. 44pp.

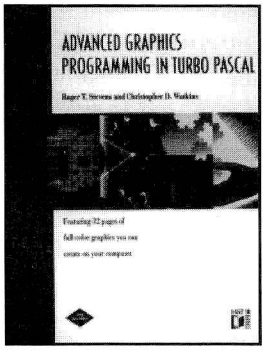
Book /Disk (MS-DOS)	**Item #118-0**	**$39.95**
Book only	**Item #134-2**	**$29.95**



Advanced Fractal Programming in C by Roger T. Stevens

Programmers who enjoyed our best-selling *Fractal Programming in C* can move on to the next level of fractal programming with this book. Included are how-to instructions for creating many different types of fractal curves. Contains 16 pages of full-color fractals. All the source code to generate the fractals is available on an optional disk in MS/PC-DOS format. 305 pp.

Book/Disk (MS-DOS)	**Item #097-4**	**$39.95**
Book only	**Item #096-6**	**$29.95**



Advanced Graphics Programming in Turbo Pascal by Roger T. Stevens and Christopher D. Watkins

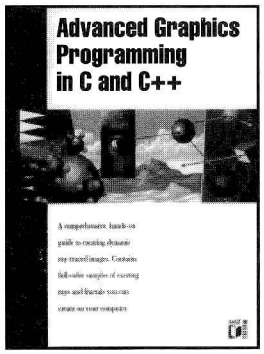
This new book is must reading for Turbo Pascal programmers who want to create impressive graphic designs on IBM PCs and compatibles. There are 32 pages of full-color graphic displays along with the source code to create these dramatic pictures. Complete explanations are provided on how to tailor the graphics to suit the programmer's need. Covered are algorithms for creating complex 2-D shapes, including lines, circles and squares; advanced 3-D shapes, wire-frame graphics, and solid images; numerous tips and techniques for varying pixel intensities to give the appearance of roundness to an object; and more. 540 pp.

Book /Disk (MS-DOS)	**Item #132-6**	**$39.95**
Book only	**Item #131-8**	**$29.95**

1-800-533-4372 (in CA 1-800-356-2002)

M&T BOOKS

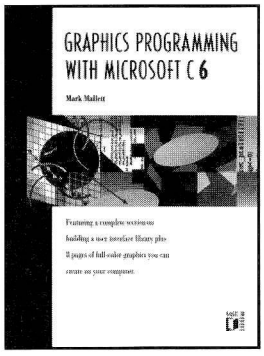
A Library of Technical References from M&T Books



Advanced Graphics Programming in C and C++ by Roger T. Stevens and Christopher D. Watkins

This book is for all C and C++ programmers who want to create impressive graphic designs on their IBM PCs or compatibles. Through in-depth discussions and numerous sample programs, readers will learn how to create advanced 3-D shapes, wire-frame graphics, solid images, and more. All source code is available on disk in MS/PC-DOS format. Contains 16 pages of full-color graphics. 500 pp. approx.

Book /Disk (MS-DOS)	**Item #173-3**	**$39.95**
Book only	**Item #171-7**	**$29.95**



Graphics Programming with Microsoft C 6 by Mark Mallett

Written for all C programmers, this book explores graphics programming with Microsoft C's built-in graphics libraries. Sample programs will help readers learn the techniques needed to create spectacular graphic designs, including 3-D figures, solid images, and more. All source code in the book is available on disk in MS/PC-DOS format. Includes 16 pages of full-color graphics. 500 pp. approx.

Book/Disk (MS-DOS)	**Item #167-9**	**$39.95**
Book only	**Item #165-2**	**$29.95**



Fractal Programming in C by Roger T. Stevens

If you are a programmer wanting to learn more about fractals, this book is for you. Learn how to create pictures that have both beauty and an underlying mathematical meaning. Included are over 50 black and white pictures and 32 full-color fractals. All source code to reproduce these pictures is provided on disk in MS-DOS format requiring an IBM PC or clone with an EGA or VGA card, a color monitor, and a Turbo C, Quick C, or Microsoft C compiler. 580 pp.

Book /Disk (MS-DOS)	**Item #038-9**	**$39.95**
Book only	**Item #037-0**	**$29.95**



Fractal Programming in Turbo Pascal by Roger T. Stevens

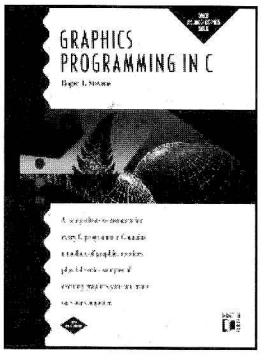
This book equips Turbo Pascal programmers with the tools needed to program dynamic fractal curves. It is a reference that gives full attention to developing the reader's understanding of various fractal curves. More than 50 black and white and 32 full-color fractals are illustrated throughout the book. All source code to reproduce the fractals is available on disk in MS/PC-DOS format requiring a PC or clone with EGA or VGA, color monitor, and Turbo Pascal 4.0 or later. 462 pp.

Book /Disk (MS-DOS)	**Item #107-5**	**$39.95**
Book only	**Item #106-7**	**$29.95**

1-800-533-4372 (in CA 1-800-356-2002)

M&T BOOKS

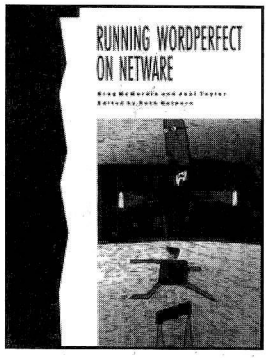
A Library of Technical References from M&T Books



Graphics Programming in C
by Roger T. Stevens

All the information you need to program graphics in C, including source code, is presented. You'll find complete discussions of ROM BIOS, VGA, EGA, and CGA inherent capabilities; improved, faster methods of displaying points on a screen; improved, faster algorithms for drawing and filling lines, rectangles, rounded polygons, ovals, circles, and arcs; graphic cursors; and much more! Both Turbo C and Microsoft C are supported. 639 pp.

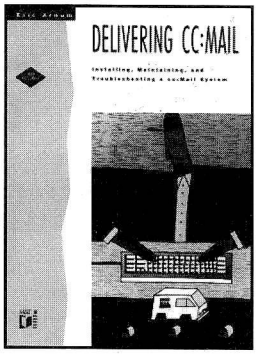
Book/Disk (MS-DOS)	**Item #019-2**	**$36.95**
Book only	**Item #018-4**	**$26.95**



Running WordPerfect on NetWare
by Greg McMurdie and Joni Taylor

Written by NetWare and WordPerfect experts, this book contains practical information for both system administrators and network WordPerfect users. Administrators will learn how to install, maintain, and troubleshoot WordPerfect on the network. Users will find answers to everyday questions such as how to print over the network, how to handle error messages, and how to use WordPerfect's tutorial on NetWare. 246 pp.

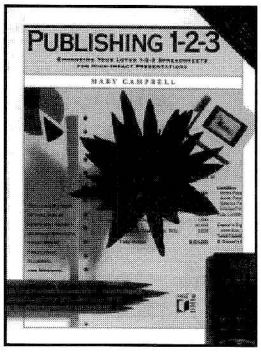
Book only	**Item #145-8**	**$29.95**



Delivering cc:Mail
Installing, Maintaining, and Troubleshooting a cc:Mail System
by Eric Arnum

Delivering cc:Mail teaches administrators how to install, troubleshoot, and maintain cc:Mail, one of the mot popular E-mail applications for the PC. In-depth discussions and practical examples show administrators how to establish and maintain the program and database files; how to create and modify the bulletin boards, mail directory, and public mailing lists; and how to diagnose and repair potential problems. Information on using the management tools included with the package, plus tips and techniques for creating efficient batch files are also included. All source code is available on disk in MS/PC-DOS format. 450 pp.

Book /Disk (MS-DOS)	**Item #187-3**	**$39.95**
Book only	**Item #185-7**	**$29.95**



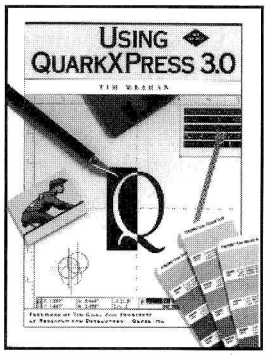
Publishing 1-2-3
by Mary Campbell

Publishing 1-2-3 teaches users how to apply their 1-2-3 skill to create winning spreadsheets and presentations. Written for users at all levels, it outlines basic design objectives and describes important elements to consider when planning a 1-2-3 spreadsheet. In addition, this book shows readers how to enhance the features of 1-2-3 by using it with different graphics and word processing packages. Readers will learn how to use graphics packages such as Harvard Graphics and Freelance to create charts and graphs beyond the capabilities of 1-2-3 and how to import 1-2-3 spreadsheets, charts, and graphs into WordPerfect®. Covers Lotus 1-2-3 versions 2.01 through 3.1. 369 pp.

Book only	**Item #199-7**	**$24.95**

1-800-533-4372 (in CA 1-800-356-2002)

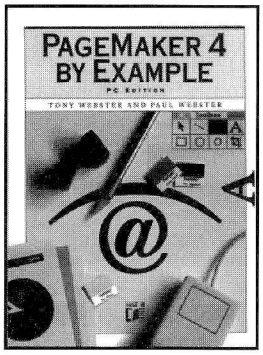
A Library of Technical References from M&T Books



Using QuarkXPress 3.0
by Tim Meehan

Written in an enjoyable, easy-to-read style, this book addresses the needs of both beginning and intermediate users. It includes numerous illustrations and screen shots that guide readers through comprehensive explanations of QuarkXPress, its potential and real-world applications. *Using QuarkXPress* contains comprehensive explanations for the concepts, practices, and uses of QuarkXPress, with sample assignments of increasing complexity that give readers actual hands-on experience using the program. Disk in Macintosh format includes exercises and templates. 340 pp.

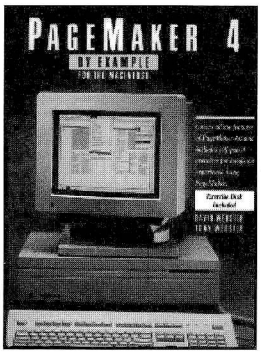
Book/Disk (Macintosh)	**Item #129-6**	**$34.95**
Book only	**Item #128-8**	**$24.95**



PageMaker 4 By Example
PC Version
by Tony Webster and Paul Webster

This hands-on tutorial introduces PC users to the new and enhanced features of PageMaker 4, providing them with a detailed look at this powerful and versatile desktop publishing program. It progressively covers more and more complex PageMaker operations, taking readers from basic functions to creating customized templates. Step-by-step exercises that teach readers how to apply the concepts learned to practical desktop publishing projects accompany each chapter. The supplementary disk includes tips and techniques for creating professional-looking newsletters, advertisements, press releases, and more. 617 pp.

Book /Disk (MS-DOS)	**Item #151-2**	**$34.95**
Book only	**Item #149-0**	**$24.95**



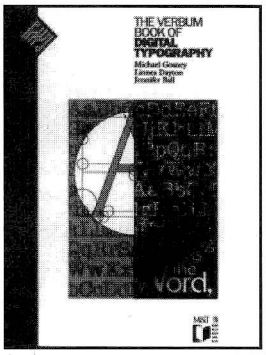
PageMaker 4 By Example
Macintosh Version
by Tony Webster and David Webster

A self-paced, hands-on guide to learning PageMaker 4.0, the premier desktop publishing program in the micro-computer market. Includes step-by-step instructions for everything from starting a new document to creating customized style sheets. Hundreds of screen shots lead readers though the concepts discussed, and useful exercises reinforce each chapter's material. Readers will learn how to load files into PageMaker, edit and manipulate text, create templates, and much more. The exercise disk (Macintosh) supplements the book's exercises and teaches readers how to apply its concepts to practical desktop publishing projects. It includes tips on creating projects such as newsletters, advertisements, and press releases. 593 pp.

Book/Disk (Macintosh)	**Item #121-0**	**$34.95**
Book only	**Item #105-9**	**$24.95**

1-800-533-4372 (in CA 1-800-356-2002)

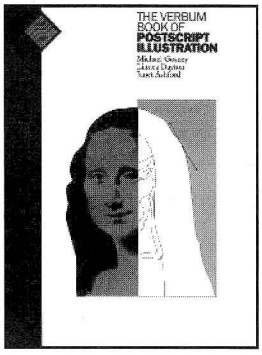
A Library of Technical References from M&T Books



The Verbum Book of Digital Typography
by Michael Gosney, Linnea Dayton, and Jennifer Ball

The Verbum Book of Digital Typography combines information on good design principles with effective typography techniques, showing designers, illustrators, and desktop publishers how to create attractive printed materials that communicate effectively. Each chapter highlights the talents of a professional type designer as he or she steps readers through an interesting real-life project. Readers will learn how to develop letterforms and typefaces, modify type outlines, and create special effects. 200 pp. approx.

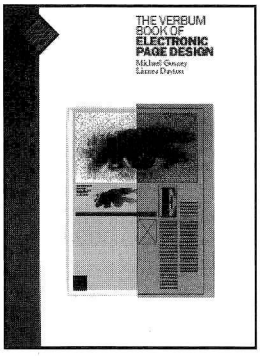
Book only **Item #092-3** **$29.95**



The Verbum Book of Postscript Illustration
by Michael Gosney, Linnea Dayton, and Janet Ashford

This is the premier instruction book for designers, illustrators, and desktop publishers using Postscript. Each chapter highlights the talents of top illustrators who demonstrate the electronic artmaking process. The narrative keys readers in to the artist's conceptual vision, providing valuable insight into the creative thought processes that go into real-world PostScript illustration projects. 213 pp.

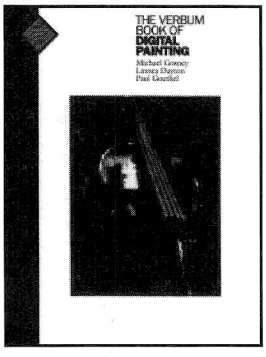
Book only **Item #089-3** **$29.95**



The Verbum Book of Electronic Page Design
by Michael Gosney and Linnea Dayton

This particular volume introduces designers, illustrators, and desktop publishers to the electronic page layout medium and various application programs, such as PageMaker, QuarkXPress, Design Studio, and Ventura Publishing. Each chapter highlights the talents of a top designer who guides readers through the thinking as well as the "mousing" that leads to creating various projects. These projects range in complexity from a trifold black and white brochure to a catalog produced with QuarkXPress. More than 100 illustrations, with 32 pages in full-color, are included. 211 pp.

Book only **Item #088-5** **$29.95**

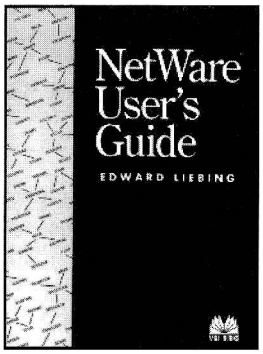


The Verbum Book of Digital Painting
by Michael Gosney, Linnea Dayton, and Paul Goethel

Contained herein are a series of entertaining projects that teach readers how to create compelling designs using the myriad graphics tools available in commercial painting programs. Presented by professional designers, these projects range from a simple greeting card to a complex street scene. This book also includes portfolios of paintings created by the featured artists, plus an extensive gallery of works from other accomplished artists and 64 pages of full-color paintings. 211 pp.

Book only **Item #090-7** **$29.95**

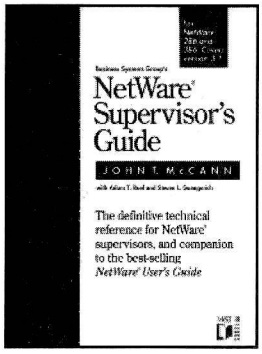
A Library of Technical References from M&T Books



NetWare User's Guide
by Edward Liebing

Endorsed by Novell, this book informs NetWare users of available services and utilities, and describes how to put them to use effectively. This book contains a complete task-oriented reference that introduces users to NetWare and guides them through the basics of NetWare menu-driven and command-line utilities. Each utility is illustrated, thus providing a visual frame of reference. You will find general information about the utilities and for performing particular tasks. The book covers NetWare v2.1 through v2.15. For advanced users, a workstation troubleshooting section describes common problems. Two appendices cover the services available in each NetWare menu or command-line utility. 520 pp. approx.

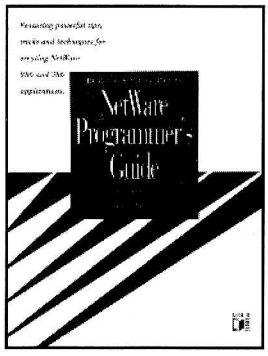
Book only **Item #071-0** **$24.95**



NetWare Supervisor's Guide
by John T. McCann, Adam T. Ruef, and Steven L. Guengerich

Written for network administrators, consultants, installers, and power users of all versions of NetWare, including NetWare 386. While other books provide information on using NetWare at a workstation level, this definitive reference focuses on how to administer NetWare. Contains numerous examples which address understanding and using NetWare's undocumented commands and utilities, implementing system fault tolerant LANs, refining installation parameters to improve network performance, and more. 510 pp.

Book only **Item #111-3** **$29.95**

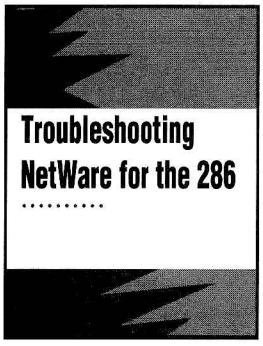


NetWare Programmer's Guide
by John T. McCann

Covered are all aspects of programming in the NetWare environment—from basic planning to complex application debugging. This book offers practical tips and tricks for creating and porting applications to NetWare. NetWare programmers developing simple applications for a single LAN or intricate programs for multi-site internetworked systems will find this book a valuable reference to have on hand. All source code is available on disk in MS-PC/DOS format. 425 pp.

Book/Disk **Item #154-7** **$44.95**

Book only **Item #152-0** **$34.95**



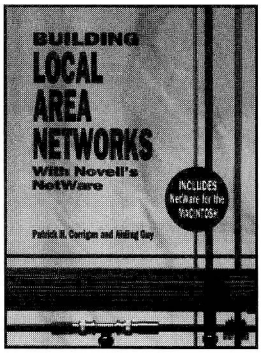
Troubleshooting NetWare for the 286
by Cheryl Snapp

The ideal reference for network supervisors responsible for the installation and maintenance of a NetWare 286 network. Contains a thorough overview of the NetWare 286 operating system plus step-by-step instructions for troubleshooting common and not-so-common problems. Detailed chapters emphasize the information most helpful in maintaining a healthy NetWare 286 LAN, including installation, file server and workstation diagnostics, printing utilities, and network management services. Covers NetWare 286 v2.2. 350 pp.

Book only **Item #169-5** **$34.95**

1-800-533-4372 (in CA 1-800-356-2002)

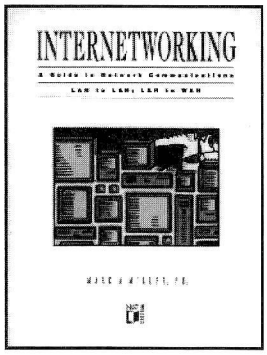
A Library of Technical References from M&T Books



Building Local Area Networks with Novell's NetWare
by Patrick H. Corrigan and Aisling Guy

From the basic components to complete network installation, here is the practical guide that PC system integrators will need to build and implement PC LANs in this rapidly growing market. The specifics of building and maintaining PC LANs, including hardware configurations, software development, cabling, selection criteria, installation, and on-going management are described in a clear "how-to" manner with numerous illustrations and sample LAN management forms. 635 pp. approx.

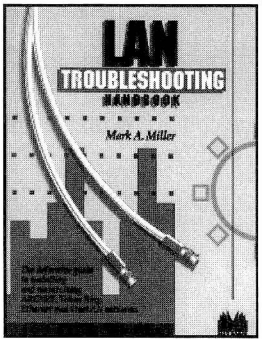
Book/Disk (MS-DOS)	**Item #025-7**	**$39.95**
Book only	**Item #010-9**	**$29.95**



Internetworking: A Guide to Network Communications LAN to LAN; LAN to WAN
by Mark E. Miller, P.E.

This book addresses all aspects of LAN and WAN (wide-area network) integrations, detailing the hardware, software, and communication products available. In-depth discussions describe the functions, design, and performance of repeaters, bridges, routers, and gateways, Communication facilities such as leased lines, T-1 circuits and access to packed switched public data networks (PSPDNs) are compared, helping LAN managers decide which is most viable for their internetwork. Also examined are the X.25, TCP/IP, and XNS protocols, as well as the internetworking capabilities and interoperability constraints of the most popular networks, including NetWare, LAN Server, 3+Open™, VINES®, and AppleTalk. 425 pp.

Book only	**Item #143-1**	**$34.95**



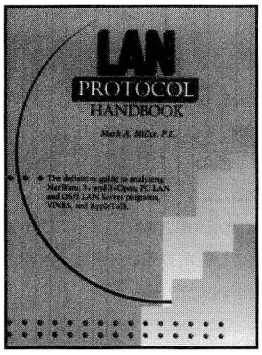
LAN Troubleshooting Handbook
by Mark A. Miller, P.E.

This book is specifically for users and administrators who need to identify problems and maintain a LAN that is already installed. Topics include LAN standards, the OSI model, network documentation, LAN text equipment, cable system testing, and more. Addressed are specific issues associated with troubleshooting the four most popular LAN architectures: ARCNET, Token Ring, Ethernet, and StarLAN. Each is closely examined to pinpoint the problems unique to its design and the hardware. Handy checklists to assist in solving each architecture's unique network difficulties are also included. 309 pp.

Book/Disk (MS-DOS)	**Item #056-7**	**$39.95**
Book only	**Item #054-0**	**$29.95**

M&T BOOKS

A Library of Technical References from M&T Books



LAN Protocol Handbook
by Mark A. Miller, P.E.

Requisite reading for all network administrators and software developers needing in-depth knowledge of the internal protocols of the most popular network software. It illustrates the techniques of protocol analysis — the step-by-step process of unraveling LAN software failures. Detailed is how Ethernet, IEEE 802.3, IEEE 802.5, and ARCNET networks transmit frames of information between workstations. Individual chapters thoroughly discuss Novell's NetWare, 3Com's 3+ and 3+Open, IBM Token-Ring related protocols, and more! 324 pp.

Book only **Item #099-0** **$34.95**

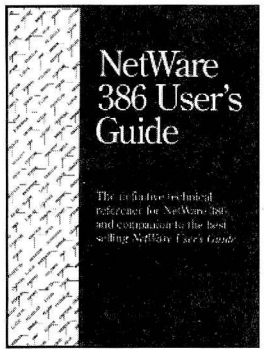
LAN Protocol Handbook Demonstration Disks

The set of seven demonstration disks is for those who wish to pursue the techniques of protocol analysis or consider the purchase of an analysis tool.

The analyzers will give you a clear view of your network so that you can better control and manage your LAN, as well as pinpoint trouble spots. The LAN Protocol Handbook demo disks are packed with detailed demonstration programs for LANanlyzer®, LAN Watch®, The Sniffer®, for Token-Ring and Ethernet, SpiderAnalyzer® 320-R for Token-Ring, and LANVista®. By surveying the demo programs, you will receive enough information to choose an analyzer that best suits your specific needs.

Requirements: IBM PC/XT/AT compatible with at least 640K after booting. Requires DOS version 2.0 or later. Either a color or monochrome display may be used.

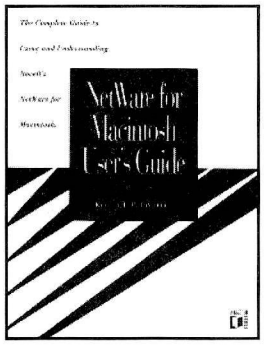
Seven Disks **Item #114-8** **$39.95**



NetWare 386 User's Guide
by Christine Milligan

NetWare 386 User's Guide is a complete guide to using and understanding Novell's NetWare 386. It is an excellent reference for 386. Detailed tutorials cover tasks such as logging in, working with directories and files, and printing over a network. Complete explanations of the basic concepts underlying NetWare 386, along with a summary of the differences between NetWare 286 and 386, are included. Advanced users will benefit from the information on managing workstation environments and the troubleshooting index that fully examines NetWare 386 error messages. 450 pp.

Book only **Item #101-6** **$29.95**



NetWare for Macintosh User's Guide
by Kelley J. P. Lindberg

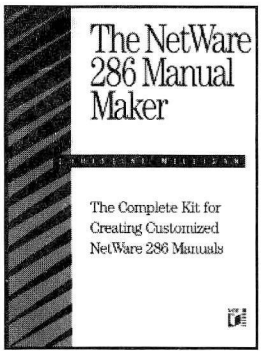
NetWare for Macintosh User's Guide is the definitive reference to using Novell's NetWare on Macintosh computers. Whether you are a novice or an advanced user, this comprehensive text provides the information readers need to get the most from their NetWare networks. It includes an overview of network operations and detailed explanations of all NetWare for Macintosh menu and command line utilities. Detailed tutorials cover such tasks as logging in, working with directories and files and printing over a network. Advanced users will benefit from the information on managing workstation environments and troubleshooting. 280 pp.

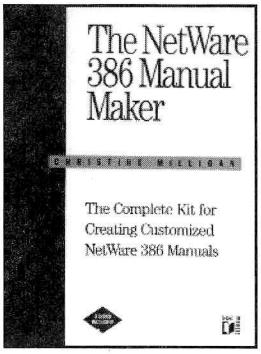
Book only **Item #126-1** **$29.95**

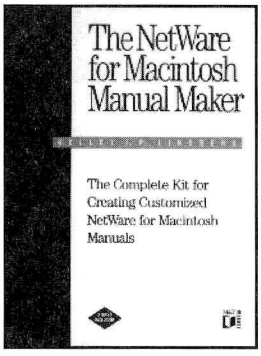
1-800-533-4372 (in CA 1-800-356-2002)

M&T BOOKS

A Library of Technical References from M&T Books







The NetWare Manual Makers
Complete Kits for Creating Customized NetWare Manuals

Developed to meet the tremendous demand for customized manuals, *The NetWare Manual Makers* enable the NetWare supervisor and administrator to create network training manuals specific to their individual sites. Administrators simply fill in the blanks on the template provided on disk and print the file to create customized manuals and command cards. Included is general "how-to" information on using a network, as well as fill-in-the-blank sections that help administrators explain and document procedures unique to a particular site. This disk files are provided in WordPerfect and ASCII format. The WordPerfect file creates a manual that looks exactly like the one in the book. The ASCII file can be imported into any desktop publishing or word processing software.

The NetWare 286 Manual Maker
The Complete Kit for Creating Customized NetWare 286 Manuals
by Christine Milligan

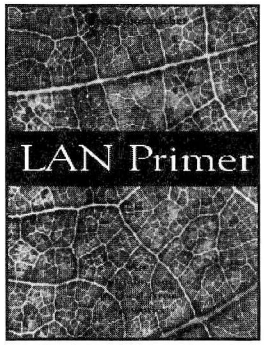
Book only **Item #119-9** **$49.95** **314 pp.**

The NetWare 386 Manual Maker
The Complete Kit for Creating Customized NetWare 386 Manuals
by Christine Milligan

Book only **Item #120-2** **$49.95** **314 pp.**

The NetWare for Macintosh Manual Maker
The Complete Kit for Creating Customized NetWare for Macintosh Manuals
by Kelley J. P. Lindberg

Book only **Item #130-X** **$49.95** **314 pp.**



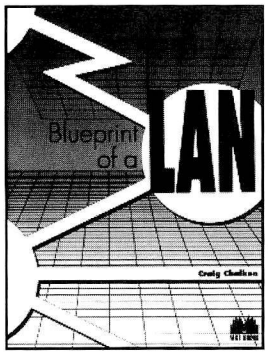
LAN Primer
An Introduction to Local Area Networks
by Greg Nunemacher

A complete introduction to local area networks (LANs), this book is a must for anyone who needs to know basic LAN principles. It includes a complete overview of LANs, clearly defining what a LAN is, the functions of a LAN, and how LANs fit into the field of telecommunications. The author discusses the specifics of building a LAN, including the required hardware and software, an overview of the types of products available, deciding what products to purchase, and assembling the pieces into a working LAN system. *LAN Primer* also includes case studies that illustrate how LAN principles work. Particular focus is given to Ethernet and Token Ring. 221 pp.

Book only **Item #127-X** **$24.95**

1-800-533-4372 (in CA 1-800-356-2002)

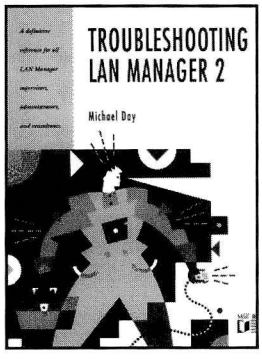
A Library of Technical References from M&T Books



Blueprint of a LAN
by Craig Chaiken

For programmers, valuable programming techniques are detailed. Network administrators will learn how to build and install LAN communication cables, configure and troubleshoot network hardware and more. Addressed are a very inexpensive zero-slot, star topology network, remote printer and file sharing, remote command execution, electronic mail, parallel processing support, high-level language support, and more. Also covered is the complete Intel 8086 assembly language source code that will help you build an inexpensive-to-install local are network. An optional disk containing all source code is available in MS/PC-DOS format. 337 pp.

Book/Disk (MS-DOS)	**Item #066-4**	**$39.95**
Book only	**Item #052-4**	**$29.95**



Troubleshooting LAN Manager 2
by Michael Day

The ideal reference for network supervisors responsible for the maintenance of a LAN Manager 2 network. *Troubleshooting LAN Manager 2* builds a functional model of LAN Manager from the ground up, beginning with OS/2 and ending with fault tolerance and printer setup. Key components such as data structures, protocols, services, and applications are placed in a troubleshooting context, examining possible problems and providing hands-on solutions. More than basic hints and tips, this book lays a solid foundation upon which you can build a truly outstanding troubleshooting methodology. 337 pp.

Book only	**Item #161-X**	**$34.95**

ORDER FORM

To Order:

Return this form with your payment to M&T Books, 501 Galveston Drive, Redwood City, CA 94063 or **call toll-free 1-800-533-4372 (in California, call 1-800-356-2002).**

ITEM #	DESCRIPTION	DISK	PRICE

Subtotal	
CA residents add sales tax ____%	
Add $3.75 per item for shipping and handling	
TOTAL	

NOTE: FREE SHIPPING ON ORDERS OF THREE OR MORE BOOKS.

Charge my:

❑ **Visa**

❑ **MasterCard**

❑ **AmExpress**

CARD NO. _____

SIGNATURE _____ EXP. DATE _____

❑ **Check enclosed, payable to M&T Books.**

NAME _____

ADDRESS _____

CITY _____

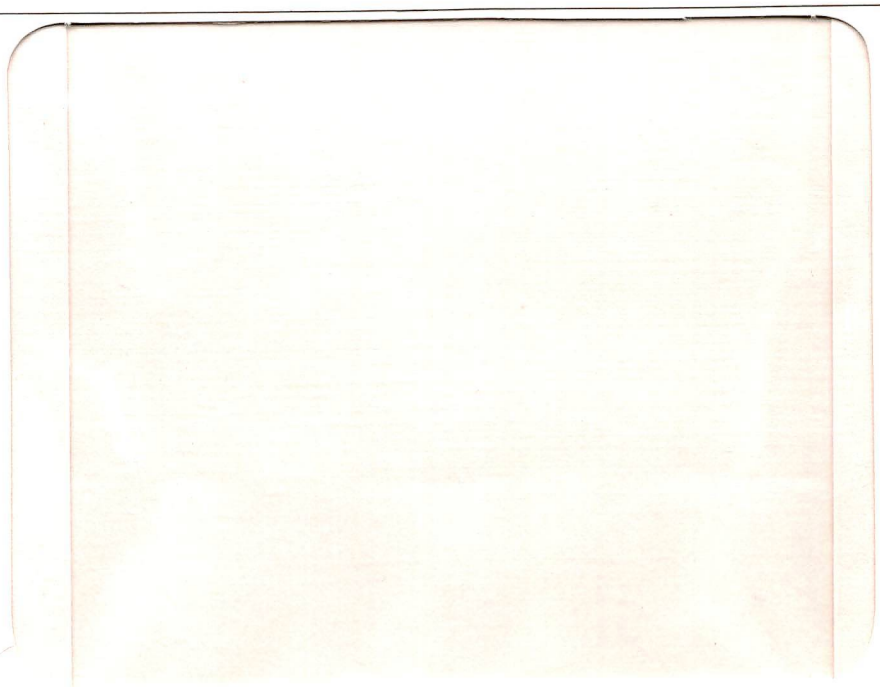
STATE _____ ZIP _____

M&T GUARANTEE: If you are not satisfied with your order for any reason, return it to us within 30 days of receipt for a full refund. Note: Refunds on disks apply only when returned with book within guarantee period. Disks damaged in transit or defective will be promptly replaced, but cannot be exchanged for a disk from a different title.

8035

1-800-533-4372 (in CA 1-800-356-2002)

CUT TO OPEN



F
D
E

Dr. Dobb's Journal Bound Volume 15 - 1990

The new decade begins and *DDJ* enters its 15th year of power programming. Every 1990 issue of *Dr. Dobb's Journal* has been collected and bound into this single volume. Included are the popular Annual C and Annual Operating Systems issues. Featured are well-known contributors such as Michael Swaine, Al Stevens, and Jeff Duntemann. There's also the *Dr. Dobb's Journal* 1989 index, and all the source code for 1990 (supplied on disk, PC/MS-DOS format). Some of the 1990 topics include:

- **Real-Time Data Acquisition.** Valuable information on all the tools you need (both hardware and software), for your own data acquisition system.
- **Three-dimensional Graphics Using The X Window System.** 3-D graphics are possible with X Window systems! Here's what can be expected from porting 3-D graphics to X, plus solutions to some thorny problems.
- **68040 Programming.** This member of the 680x0 family provides challenges for programmers at all levels.
- **Neural Nets.** DDJ presents an environment that dynamically creates neural networks. Also included are discussions of the similarities and differences of various neural net models.
- **Memory Management.** Everything from how to take advantage of "handle pointers" to object swapping.
- **Hypertext.** A behind-the-scenes look at the *DDJ* hypertext project.
- **Graphics.** From Super VGA programming to drawing character shapes with Bezier curves.
- **C Programming.** Porting C programs to 80386 protected mode, encapsulating C memory allocation, parallel extensions to C, and much more!
- **Unraveling Optimization.** Examined are the practical and theoretical aspects of code optimization using Microsoft C 6.0.
- **Communications & Connectivity.** Controlling Unix processes, designing for OSI, and programming with Mac Comm toolbox.

53995
9 781558 511835
ISBN 1-55851-183-0
>$39.95

M&T BOOKS

M&T Publishing, Inc.
501 Galveston Drive
Redwood City, CA 94063