As we noted last issue (II, 2), Commodore now offers a package containing 1) all SuperPET manuals, including COBOL, 2) replacement pages for all manuals to update them to Version 1.1, and 3) Version 1.1 disk software for all languages, including COBOL. Price is $49.95. Order Part Number 900030, "SuperPET Manuals and Languages." Either dealers or individuals may order the package (don't ask for parts of it) from Commodore Computer Systems Division, Parts Dept, 1200 Wilson Drive, West Chester, PA 19380. We're sorry to report that the second package reported last issue (COBOL manual, manual update pages and V1.1 software) is not available. If any of you need V1.1 manuals or software, go get it!

NEW COMPUTERS A few months back, Commodore bought the rights to the Amiga computer; this 68000 machine should come to market in 1985. The boss of Commodore is a veteran of the clothing trade; we wonder if it'll be cut on the bias: cheap, reliable, bereft of any shred of customer support, and closed to all outside software vendors, in the good old Commodore tradition. We hope not. We also read that kindly Uncle Jack Tramiel, the new boss of Atari, plans to issue new 8-, 16-, and 32-bit computers in '85, assisted by his old Commodore management team, now at Atari (is that good news or bad news?). Seems that Digital Research has written an operating system overlay for the new line--GEM, with pull-down menus, icons, windows, and all that MacJazz. The new computers are supposed to sell for less than $1000. Maybe we'll see some options between the IBM Scylla and the Apple Charybdis. And maybe not.

In the latest MIDNIGHT/PAPER, Jim Strasma tells us that Commodore has cut off most of its remaining U.S. dealers, leaving about 40 in the country. Even if Commodore produces a gee-whiz Amiga with a 68000 microprocessor, who'll want one without any dealer support? (Anybody consider Toys R Us and K-Mart support?) We also laugh (hysterically) at the thought of the local toy dealer demonstrating MS DOS on Commodore's rumored PC clone. Or does Commodore plan to somehow find shelf space at ComputerWorld and the other chains--at a time when those stores are winnowing out the losing clones and wondering when the computer market will recover? With its fortunes tied to the aging C-64, we suspect Commodore is headed for big trouble; Atari and kindly Uncle Jack are, of course, already in it. (We swiped "kindly Uncle Jack" from its inventor, kindly Hal Hardenbergh.)

IS YOUR ADDRESS LABEL REDMARKED?

If so, check the RENEW block on the last page and send it with your address label or a copy (don't fill out the form if you send the label). Remit $15 U.S. in North America or $25 if elsewhere. Please do it before Old Weakeyes must scratch your file and then (ugh!) retype the whole address. Make checks to ISPUG. Mark all letter bombs clearly so we may evacuate the innocent.

PRICE CUT ON COM-MASTER; ISPUG DISTRIBUTES IT When first we reviewed the terminal emulator COM-MASTER, written by ISPUG member Dan Jeffers out in Hawaii, we called it a fine telecommunications program. Since then, Dan has adapted the program to handle data in APL as well as it does in the other languages; Steve Zeller, in his last review of it, called it the best terminal program existing for SuperPET, bar none. We've since had confirmation of that view from Barry Bogart and a few others. Because Dan is a professional programmer with other fish to fry, he's asked ISPUG to distribute COM-MASTER henceforth at a price of $50 per copy of disk and manual. We agreed.

Reviews of COM-MASTER are found in the Gazette in I, 136, 210, 222, 269. The program receives and transmits both SEQ and PRG files, and can handle 5, 6, 7,

and 8-bit codes; incorporates XON and XOFF protocol if you need it, and lets you store, on disk, files which will configure COM-MASTER as you want it for any specific application. Copies may be ordered from the editor, at PO Box 411, Hatteras, N.C. 27943. State 4040 or 8050 format! Disks are not copy-protected; you may make your own backups. This is a 6809-side program in machine language; it'll handle files created in both 6809 and 6502 sides of SuperPET.

A PATCH FOR MFORTRAN'S NEGATIVE INTEGER READ BUG

Last issue, Associate Editor Stan Brockman reported on the bug in READing negative integers from a disk file in microFORTRAN. Shortly after we went to press, WATCOM sent a patch for it. Happily, WATCOM adopted Associate Editor Terry Peterson's superfast patch program, of which we wrote in I, 266. WATCOM's old patches ran in half an hour or so; this one finishes the job in a bit over three minutes. Beware a bad version of this patch, in which line three of the DATA statements begins: 141,21,237... It's wrong. Use the patch below.

```
120 ! mFORTRAN patch 3, title: for_patch3:bp. Patch V1.1 only.
130
140 t0=time
150 dim a%(46)
160 data 71,25,52,22,221,152,227,102,237,98
170 data 236,102,141,170,227,228,237,228,236,98,52,6,48,98,31,16
180 data 141,19,237,248,10,236,98,163,100,38,3,198,255,33,95,29
190 data 50,102,57,-1,-1
200 open #2,"(f:128)disk/1.FORTRAN,PRG",input   ! Wish (f:512) allowed.
210 x = peek(86)*256 + peek(87) + 4             ! Let's fake it!
220 y = peek(x)*256 + peek(x+1) + 1             ! We'll find LRecL and
230 poke y,2,0                                  ! tell it to be 512 bytes.
240 open #3,"disk/0.FORTRAN,PRG",output
250 x = peek(86)*256 + peek(87) + 4       ! Make LRecL 0 for output file.
260 y = peek(x)*256 + peek(x+1) + 1
270 poke y,0,0
280 mat read a%
290 i%=0 : j%=1 : LRecL%=512 : CR$=chr$(13)
300
310 loop
320   for j% = j% to a%(i%)     ! Get 71 chunks of 512 bytes of code (see 71
330       call GetRec           ! in first DATA statement). Transfer them un-
340       print #3,l$;          ! changed to the new disk file.
350   next j%
360   i%=i%+1
370   call GetRec                 ! Get offending 512-byte record.
380   for k%=a%(i%)+1 to LRecL%   ! Start with byte 26 (see 25 in 2nd DATA
390     i%=i%+1                   ! statement), and amend program from DATA
400     if a%(i%)=-1 then quit    ! statements until DATA is -1.
410     l$(k%:k%)=chr$(a%(i%))
420   next k%
430   i%=i%+1 : print #3,l$;      ! This program is designed to handle multiple
440   j% = j% + 1                 ! patches; in this case, we have only one,
450 until (a%(i%)=-1)             ! and so quit on the final -1 in DATA.
460 on eof ignore
470 loop
480   call GetRec                 ! Thereafter, we copy the remainder of the
490   Stat_2 = io_status          ! old program unchanged.
```

```
500   print #3,1$;
510 until Stat_2
520 close #2 : close #3
530 print time-t0
540 stop
560
570 proc GetRec
580   linput #2,1$
590   while len(1$)<LRecL%
600     linput #2,11$
610     1$=1$+CR$+11$
620   until io_status
630 endproc
```

We recommend you place a copy of mFORTRAN, as patched with mFORTRAN patch 2 (I, p. 208), on drive 1, and put your master language disk in drive 0 (we trust you have a backup of it!).

Then run the patch. If you have copied accurately, the patched version will properly read negative integers from a disk file. After you test the new version thoroughly, replace all current mFORTRAN copies with the patched one. Save an unpatched copy of mFOR (bugs later?). For mFORTRANners not familiar with mBASIC: 1) Be sure to include all terminal semicolons in any program line, 2) you may in mBASIC get the mED, with which you're familiar, with the command: edit <RETURN>. Once you have it, it will work exactly as does mED in mFORTRAN, including a RUN with SHIFT/RUN.

ONCE OVER LIGHTLY
Miscellany

Did you know that if you invoke SETUP from the monitor or from main menu, you reset SuperPET's interrupt table at $0108 to its default or bootup values? This is deadly. If, for example, you have an interrupt-driven ML module in memory, it will not execute after you refer to SETUP. The address of your interrupt-driven routine is wiped out. Any interrupt-driven routine (PIRQ, UDUMP, most telecom programs) is affected. Lesson: use SETUP before you load any interrupt-driven module; or, if you invoke SETUP in a program, reset the proper address for your interrupt routine at $108. We learned this from Brad Bjorndahl of Bramalea, Ontario, who also notes that MemEnd_ isn't reset to $7FFF by SETUP (We suspect young Bodsworth did this; who else would reset one bootup value and leave the other alone--and not tell anybody?).

A 68000 CROSS-ASSEMBLER Grand satchem Terry Peterson has done it again! We wrote last issue about HALGOL, a new language designed to run on the Motorola 68000 microprocessor (and about a black box which'll soon hitch up to the 6502 side of SPET to put both to work). Seems Terry has a prototype of the black box and an early version of HALGOL; to get at the 68000 itself, he needed an assembler to generate 68000 code, and then proceeded to modify one for all Commodore machines. ASSEM68K, the Phase Zero, Ltd. 6502-to-68000 Cross Assembler, now runs on the 8032 or 4032 (with 4040, 8050, or 8250 drives) and on the C-64 with a 1541 drive. Written in 6502 machine language, it cross-assembles 68000 source code into 68000 machine language, saved to disk in binary. ASSEM68K is supplied with a 15+ page manual and a sample loader program. The loader reads the object files on disk and transfers them to a 68000 attached-processor board (the DTACK GROUNDED/Grande). You can prepare the source files on 1) The Commodore Assembler Development System Editor, 2) any word processor able to write ASCII or PETASCII files, or 3) on SPET's microEDITOR (a version of which now runs on the C-64). Price: $95 U.S., check or postal money order, from T&S Peterson Software Products, 8628 Edgehill Court, El Cerrito, CA 94530. Add 6.5% sales tax if you survived in California after the reign of King Fuzzwump Brown and courtiers.

ON COPYING DISK FILES We've received a lot of letters about boo-boos in the COPY command in the mED. See page 53 of the System Overview manual (V1.1).

The first difficulty: some folks don't realize that the DOS format of the file (SEQ, PRG, USR, REL) must be specified as part of the COPY command. There is,

however, a default to SEQ. In the example below, left, a SEQ file 'example' is copied from drive 0 to drive 1. If that same file should be a PRG, USR, or REL file, the command at left will fail to a 'file type mismatch' error. The DOS format (for other than SEQ files) must be specified in the COPY command, as shown in the next example at left for a PRG file. Which leads us to the booby-trap: if you forget to include the DOS format ('prg,' as shown in the example) when you specify the destination filename, COPY will indeed copy the file--but as a SEQ file! And, of course, when you try to load or run it, the mis-formatted copy fails.

```
copy example to disk/1.example
```

```
copy example,prg to disk/1.example,prg
```

In contrast, the 3.0 DOS commands, prefaced by "g ieee8-15.", and given at command cursor in the microEDITOR, will copy any file of any format properly from one disk to another. If you use "g ieee9-15", the commands work on device 9. You need not specify the DOS format. The proper one is selected by the DOS. The code shown at left is bug-free and simple; it will always copy to the destination drive the designated file in original format--faster and with less typing. The RENAME and SCRATCH commands don't share the flaw; you needn't specify DOS format. Unfortunately, COPY is the only command available in 6809 to copy files between devices 8 and 9.

```
[destination always left of =]
g ieee8-15.C1:*=0:example
```

COPY TO TERMINAL Gee, we got a complaint that you can't read any files in mED if you have a file on screen. Not so, Bodsworth. Say at command cursor: copy filename to terminal; thee will see the whole file--even if it scrolls by pretty fast. Touch STOP to stop the load at any time. Carry on, Bodsworth.

BELOW AVERAGE! By definition, half the population of the globe is below average in intelligence, however measured. We wonder why the lower half seems to be in charge of naming things. Noticed a new sign on a county building the other day: Jail Facility Building. Whoever is responsible ought to be thrown in it. A jail, says Webster, is a building for confinement. How about plain "Jail"? That one isn't as bad as the new signs on our police cars: Security Police. Ever hear of Insecurity Police? Then we read about some basic fundamentals in TIME (strain the basics from the fundamentals, hmmm?), while the TV parson told us to "gather together" to pray. Whilst we pondered how to gather untogether, he spoke of some fundamental interrelationships (obviously not related?). Then we saw a User Instruction Manual. Gee, do they write 'em without instructions for non-users?

THANK YOU, DR. DOBBS In the December '84 issue of this good, grey magazine, we ran into a handy-dandy toggle, which cycles always between the values of one and zero. Write: toggle=1-toggle in a loop. See what happens? Elegant.

DEVELOPMENT MANUAL ERRORS Chuck Robinson of Minneapolis sent some notes on changes you should mark in this book: On page 162, the MODIFY command (which changes specified sections of memory) should be entered as: >m 00 00 00, with spaces between entries, not commas as shown. On page 173, any string converted by STOI_ cannot hold any leading spaces; on page 182, library program DIRCLOSEF_ should be titled DIRCLOSE_, as it's shown on watlib.exp on disk. He also adds, for those interested in driving a monitor from SuperPET, that pin 2 of the User Port is TTL Video, pin 9 is TV horizontal, and pin 10 TV vertical--the latter two presumably also TTL. He says the TTL level may be low for some monitors.

USER INDEX TO HAYES SMARTMODEM MANUAL Frank Brewster, 1 North Vista Avenue, Bradford, PA 16701, writes that he recently got a Hayes Smartmodem and User Manual, written in High Abyssinian Greek (all computer engineers speak this language; ordinary folk require a four-phase Gaines-Moellner Transfinite Encabulator to decrypt it). After two weeks or so, Frank finally wrote an index which groups all references to whatever function he wants to perform whilst tc'ing. He offers it to any equally frustrated member of ISPUG who also misses a functional index in the Hayes manual. (Those who publish a manual with no index should, as penance, translate Webster's unabridged from Swahili into Urdu.) Send a SASE to Frank (a big one, size 10, dammit; those tiny 6.5 x 3.5 inch envelopes we get won't hold a brief mash note to an elf!).

BLACK MAGIC ON THE SERIAL PORT Friend Frank (above) also reports mysterious stuff when his printer is on the serial port. On stray occasions, turning the printer on or off crashes SPET, even when the printer file is not open. When his modem is on the serial port, turning it off and on can do the same thing. Even more surprising: with his printer on the IEEE-488, and using his modem, a disconnect from Telenet, during TC, can throw his printer into graphics mode, even though the printer file is not open. We hesitate to mention the next one: if his printer is on the serial port, it linefeeds with every CR--with the printer's switch for linefeed OFF. On the IEEE-488, the printer linefeed switch must be on to get ary a linefeed with a CR. Any wizards out there who can deal with this? Or is his serial port bewitched? Send advice or powdered eye of newt.

Loch Rose reports similar problems: his parallel printer attaches to an ADA 1800 interface on the IEEE-488. If he uses disks and computer for a while, then turns the printer on and does a disk operation, the printer prints a DOS command, such as "$0" for a directory on drive 0--but only on the first disk operation after printer ON--in both 6809 and 6502. Hmmm. Last, if he tries to use his printer with the disk drive switched off, printer speed is about one-quarter normal--sometimes. We have a printer cabled directly to the IEEE at one disk drive, and it works fine--unless the printer is on when the computer is turned on, which crashes SPET. When we 1) turn on computer 2) turn on disk drives 3) turn on the printer, and 4) load what we want, everything is fine. Any explanations for the weird events described above are welcome! Gurus, please respond.

CHECKING THOSE ROMS We stuffed disk images of all SPET's ROMs on the ISPUG Utility Disk so folks could see if they had bombed. SPMON, Terry Peterson's extended monitor, lets you compare the disk images with the ROMs themselves. We carefully gave an example in the SPMON instructions of how to do it--but not carefully enough. We loaded the disk images into memory at $6000, and then ran the COMPARE instruction. Poor fella called and said he crashed doing this, time after time. No wonder. SPMON loads at $6000, and the ROM images overwrote it. We ran our tests using SPMONLO, which loads at $2000. There are two versions of SPMON, one loading high and the other low; they were created to avoid just such conflicts in address. Both are on the ISPUG Utility Disk. Use the right one.

OOPS, AHEM, AND ON YOUR TOES DEPT. In I,121, in an opus on SuperPET files, we carried on at length on how to form filenames. Seems that you designate a non-default filetype (TEXT is default in all but APL, where VARiable is default) on drive 0 with: (v)example,usr, for a VAR file--or (f:120)example,rel for REL. Then we said to form such files on drive 1 with: disk/1.(v)example,usr. Wooops! Nobody caught that one in over 14 months.... It's wrong. Old Weakeyes blew it. The proper format: (v)disk/1.example,usr. The filetype, whether Variable, Text,

or Fixed, always precedes the device. Please change page 121, Vol. I, and Refsheet 4, p.2 of the Starter-Pak manual. Stop giggling, Bodsworth.

THE APL EXPRESS

by REG BECK
Box 16, Glen Drive, Fox Mountain, RR#2, Williams Lake, B.C., Canada V2G 2P2

First, we'll look at the QUAD functions for disk I/O operations. We'll stick to APL-seqential and BARE-sequential files and leave relative files for another column. It is worthwhile, before beginning, to read chapters 4 and 5 of the SuperPET System Overview manual to understand the different filetypes (Text, Variable and Fixed). In APL, the Variable file is the default filetype for disk I/O. Other file types need to be referenced according to the rules in the System Overview manual and those on page 90 of the APL manual.

In the definitions following, please distinguish between the order in which data are filed or retrieved (sequentially or at random), the Waterloo filetypes described above, and the DOS formats of SEQ, USR, PRG and REL, which appear on a directory. All SEQ, USR, and PRG files store data sequentially, whatever the Waterloo filetype. Only Fixed files, in REL format, accessed at random, do not store data in the sequence filed. If you are fuzzy on the differences between data order, Waterloo filetype, and DOS formats, refer to the article on SuperPET files in I, 117 (Issue 9).

As in the last column, we will stick to device #8 in the following discussion. The specific functions and definitions are found in chapter 9 of the SuperPET APL manual. Some definitions follow:

term	definition
APL value	A matrix or vector, either numeric or character, or a number.
record	An individual data item consisting of an APL value of any shape, rank and type.
file	A collection of records or 8-bit binary characters external to the active workspace.
sequential	Data are stored in the sequence filed and usually appear on directory as SEQ files, though they may also be PRG files. All Waterloo filetypes (Text, Fixed, and Variable) may be used according to the rules in System Overview.
APL-sequential file	APL values are stored in sequential order. These files default to VAR,SEQ files. They may be formed as TEXT,SEQ files. Each value is treated as a separate record in the file, and when retrieved takes the exact form it had when filed. The files contain data about the structure of the APL value as well as the value itself. Text is treated as a character vector. The first eight bytes of the file are data bytes.
BARE-sequential file	8-bit binary characters are stored sequentially in files which default to VAR,SEQ, though they may be formed as TEXT, SEQ files. APL character values are stored strictly as bytes in the sequence filed. No structural information is stored with them. When retrieved from file, they are stored in workspace as a character vector. Each sequence of characters separately filed or appended constitutes a record. In VAR,SEQ format, the first file byte shows as NULL; such files cannot be read in the mED. Files in TEXT,SEQ form can be so read.
Relative file	A sequence of 8-bit binary characters stored as groups of fixed size records (Waterloo Fixed filetype). They are accessed in

random order and will be covered in later columns; noted for complete definition. Seen on directory as REL files.

Program file A special type of BARE-sequential file which stores workspaces. Accessed using LOAD, COPY and PCOPY system commands and functions; of Variable filetype, and PRG in DOS format on directory.

tie-number An integer scalar used to reference an active file. Positive integers to 32767 are permitted as ties.

tied The active workspace and the disk drive are connected with respect to the named file by a tie number.

untied A previously active file is disconnnected.

file-designator Up to 15 characters, including punctuation (but no blanks), assigned according to rules in System Overview as the title or the name of a file.

filename A combination of device and file-designator, as in disk/1.example. Often mis-used to mean the file-designator.

---

1. General functions-- These are used for both APL-sequential and BARE-sequential files. Obviously, any file must be first created on disk before it can be accessed for input. To create a file, you must assign a filename and tie it, at which time you may send data to it. Once all data are sent, the file must be untied. See examples below and in parts 2 and 3.

Before data are received from a file on disk, the filename must be tied. The received data from the tied file is assigned to an appropriate variable; when the receive operation is finished the file is untied. See examples below.

You may create an empty file if you assign a name to a tied file and then untie it without sending any data. If you attempt to read something from such a file and assign it to a variable, the variable is found to be empty.

Whenever a filename is referenced in functions it must be placed in quotes, or, more precisely, within apostrophes (the shifted k on the APL keyboard).

```
'EX' ⎕CREATE 1          ⍝A FILE NAMED EX IS CREATED AND TIED WITH A 1.
     ⎕UNTIE 1           ⍝EX IS UNTIED.  EX IS ALSO EMPTY AS NOTHING
                        ⍝WAS WRITTEN TO IT.

       ⎕ERASE 'EX'      ⍝ERASES THE NAMED FILE, EX.  ONLY 1 FILE AT A
                        ⍝TIME MAY BE ERASED.  WORKS WITH ALL SEQ. FILES.
'EXAMPLE' ⎕RENAME 'EX'  ⍝FILENAME CHANGED FROM EX TO EXAMPLE.
       ⎕STATUS          ⍝HAS AS ITS VALUE I/O ERROR MESSAGES RELATING
                        ⍝TO THE MOST RECENT I/O OPERATION.
```

2. APL-sequential files-- Records are stored sequentially. Each record is read in using a separate read statement, or in a loop. In any one file, each record may differ in shape, rank and type. When read in, the value is assigned to a variable which takes on the appropriate shape, rank, and type.

```
MAT←10 10⍴⍳100          ⍝THE MATRIX, MAT, IS DEFINED.
'MAT1' ⎕CREATE 5        ⍝A VAR. LENGTH APL×SEQ. FILE IS CREATED.
   MAT ⎕WRITE 5         ⍝MAT IS SENT TO THE FILE, MAT1.
       ⎕UNTIE 5         ⍝MAT1 IS UNTIED.
```

In the above example only one record is filed. It may be read out again and may

be assigned to any variable name. A second read operation attempted directly after the first will yield an empty value. The file formed is VAR,SEQ.

```
'MAT1' ⎕TIE 2          ⍝THE EXISTING FILE MAT1 IS ACCESSED FOR READING.
     K ⎕READ 2         ⍝A RECORD IS READ FROM MAT1 AND PLACED IN K.
       ⎕UNTIE 2        ⍝MAT1 IS UNTIED.
```

The matrix M which was filed in MAT1 by the initial write operation is read back in and assigned to the variable, K.

```
A←5.5                  ⍝A SCALAR, A, IS DEFINED.
B←⍳25                  ⍝A VECTOR, B, IS DEFINED.
'MAT1' ⎕APPEND 3       ⍝MAT1 IS ACCESSED TO ADD MORE RECORDS.
     A ⎕WRITE 3        ⍝A IS WRITTEN TO MAT1.
     B ⎕WRITE 3        ⍝B IS WRITTEN TO MAT1.
       ⎕UNTIE 3        ⍝MAT1 IS UNTIED.
```

The above method is used to add records to an existing file. The records are appended sequentially after any already in the file.

```
VECT←'RUMPLESTILTSKIN'  ⍝A CHARACTER VECTOR IS DEFINED.
'(T)CHARS' ⎕CREATE 1    ⍝A TEXT FILE NAMED CHARS IS CREATED AND TIED.
      VECT ⎕WRITE 1     ⍝THE CHARACTER VECTOR IS SENT TO CHARS.
           ⎕UNTIE 1     ⍝THE FILE IS UNTIED.
```

In this example a TEXT file is created by prefixing the filename with (t). Each record sent could be a character vector or an entire character matrix. The file formed is a TEXT,SEQ file.

Only the quad READ and quad WRITE functions are used for sending and receiving data when accessing APL-sequential files.

3. BARE-sequential files-- All data are stored in byte form. One or more records may be sent to the file using a single quad PUT statement. To read the data requires one quad GET statement for each record in the file. The quad GET statement also stipulates how many bytes of the record will be read. Any remaining in that record will not be received by the next quad GET statement, which will input bytes from the next record. The file formed is of VAR,SEQ format.

```
'TEXT.EX' ⎕CREATE 4     ⍝FILE CREATED AND TIED.
  'ABCDE' ⎕PUT 4        ⍝CHARACTER VECTOR 'ABCDE' SENT TO FILE.
          ⎕UNTIE 4      ⍝FILE UNTIED.

'TEXT.EX' ⎕TIE 1        ⍝FILE ACCESSED.
       V← ⎕GET 1 5      ⍝5 CHARACTERS BROUGHT IN AND ASSIGNED TO V.
          ⎕UNTIE 1      ⍝FILE UNTIED.
```

Only character data may be stored in bare sequential files. The data may be sent as character vectors or matrices. Each vector or row of a matrix will be stored as one record. The data may also be sent using atomic vectors, as in the following example. This method is useful for sending unprintable characters to a file.

```
        'EX' ⎕CREATE 1  ⍝FILE CREATED.
⎕AV[1 5 6 9] ⎕PUT 1     ⍝DATA SENT.
             ⎕UNTIE 1   ⍝FILE UNTIED.
```

When unprintable characters are received from a file they can be examined using atomic vectors. In the following example, the code number corresponding to the received character is printed.

```
'EX' ⎕TIE 1              ⍝FILE ACCESSED.
 ⎕AV⍳⎕GET 1 3            ⍝BRINGS IN 3 BYTES AND DISPLAYS THEIR CODES.
     ⎕UNTIE 1            ⍝FILE UNTIED.
```

Bare sequential files are often used to output data to printers and modems. In this case, the filename contains a reference to the device. Note that you PUT and GET data in BARE-sequential files.

It is interesting to play around with both APL-sequential and BARE-sequential files to see if there is any benefit obtained from using one or the other type. For instance, a large character vector sent to disk using an APL-sequential variable length structure, an APL-sequential text structure and a BARE-sequential file took up the same 23 blocks of space in each case. Through this kind of experimentation you can obtain a more thorough understanding of how these file structures work.

Last year, I received a book titled, BASIC for Microcomputers: Apple, TRS-80, PET, by Haigh and Radford. One chapter is devoted to models and simulations. The examples and discussion are so interesting; I would recommend that anyone with an interest in these topics should pick up a copy. The excellent development of pseudocode for teaching BASIC is a further inducement. The Leontief model of an economy, a model of a pasture ecosystem, Okun's Law simulation, a genetics simulation and a Monte Carlo inventory simulation are covered. An exercise set includes a matrix model of a food web and a communications network. The BASIC code is easily rewritten in APL, which is a natural for this topic.

In the Leontief model, matrices (and vectors in APL) are used to determine the gross production of commodities given the internal consumption of these commodities and the required export amounts. I'll work through an example of the Leontief model. First, the matrix of commodities (called the technological matrix, T) is defined:

```
      steel  timber  beef  coal  transportation
T =     0.1     0.0   0.0   0.3            0.1   steel
        0.0     0.0   0.0   0.2            0.2   timber
        0.0     0.0   0.0   0.1            0.1   beef
        0.1     0.1   0.0   0.1            0.2   coal
        0.3     0.1   0.1   0.2            0.0   transportation
```

Here, the elements in the transportation row and the steel column mean that it takes .3 units of steel to produce 1 unit of transportation, and that .1 units of timber, .1 unit of beef and .2 units of coal are also required. There are similar interpretations for the other rows. The export matrix is a vector of the net amount, in units, of each commodity to be exported. The example at left means that we wish to export 3000 units of steel, 5000 units of timber, and 800 units of beef, but no coal or transportation.

```
E = (3000 5000 800 0 0)
```

What we need to know is the gross production of each commodity in order to meet the requirements of internal consumption and export. The gross production vector

is the solution to the problem and is found by subtracting T from the identity matrix, inverting the result and multiplying E by it. The solution for our example in units, is shown at left, below. I have a complete workspace for the Leontief model but the functions for matrix entry and display are too long to print here. An abbreviated function for processing the matrices is given below. The technological matrix and export vector are entered into the workspace as global variables. The number of commodities, A, must also be entered.

```
G = (4487.67 5570.88 1034.01 3368.78 2340.10)
```

```
A←5                          ⍝5 COMMODITIES.
E←3000 5000 800 0 0          ⍝EXPORT VECTOR AND TECHNOLOGICAL MATRIX.
T←5 5⍴.1 0 0 .3 .1 0 0 0 .2 .2 0 0 0 .1 .1 .1 .1 0 .1 .2 .3 .1 .1 .2 0
```

```
      ∇LEON[⎕]∇
[  0]    LEON ;TT;G
[  1]   TT←⌹((⍳A)∘.=⍳A)-T
[  2]   G←(⌊((E+.×TT)×100))÷100
[  3]   ⎕TC[5]
[  4]   ⍴1
[  5]   'THE GROSS PRODUCTION MATRIX IS---'
[  6]   ⍴1
[  7]   ▼G
```

Lines 1 and 2 of LEON replace a couple of pages of nested FOR-NEXT loops in the original BASIC program. In line 1 the identity matrix is generated, T is subtracted from it, and the result is inverted using quad divide. In line 2 the gross production matrix is calculated and rounded to 2 decimal places. To try this example, define LEON first, enter A, T, and E, and then return LEON.

The screen dump to disk on page I,109 of the Gazette is very useful in creating text files on disk from material on the screen. The APL text may then printed with ASCII text using a suitable direct dump from disk to printer. ADUMP, which is available on the ISPUG Utility disk, works fine. [Ed. I just change printwheels if in the mED and "copy filename to ieee4"] Your printer must, of course be converted to APL before printing the APL text. DTODISK, on this page, is a modified version of the previous dump--which used no loops but sent an extra carriage return, which always entered a final, blank line in the file. When you dump to disk, make sure the material to be filed begins on the very first screen line. Return DTODISK 'filename' on the line just below the last file line to be dumped. No extra blank lines will be formed if you do it this way. DTODISK appends .TXT to the end of the filename. Text files saved in this way can only be edited by bringing them back to the APL screen and redumping after editing. Also, as Steve Zeller pointed out, reduce the length of long screen lines by two characters for every overstruck character in that line or they may exceed 80 characters and be truncated.

FETCH 'filename' will bring back the text for editing in APL. It also appends .TXT to the filename. Line 10 in FETCH homes the cursor and keeps it there so that the text starts from the first screen line.

```
      ∇DTODISK[⎕]∇
[  0]    DTODISK NAME ;FILE;ROW;⎕IO;SCR;I
[  1]   FILE←'(T)',NAME,'.TXT'
[  2]   FILE ⎕CREATE 1 + ⎕IO←0
```

```
[  3]   ROW←¯1+1↑(256 256⊤⎕SYS 45188)-1
[  4]   SCR←((ROW,80)⍴⎕PEEK 32768+(80×⍳ROW)∘.+0 79)
[  5]   ⎕IO←1+I←0
[  6]   LOOP:     (⎕XR SCR[I←I+1;]) ⎕PUT 1
[  7]             →(I≠ROW)/LOOP
[  8]   ENDLOOP: ⎕UNTIE 1
      ∇FETCH[⎕]∇
[  0]    FETCH NAME ;⎕IO;I;A
[  1]   NAME←'(T)',NAME,'.TXT'
[  2]   ⎕IO←1+I←0
[  3]   M←25 79⍴' '
[  4]   NAME ⎕TIE 1
[  5]   LOOP:     M[I←I+1;]←A,(79-⍴A←⎕GET 1 79)⍴' '
[  6]             →LOOP×⍳((+/'EOF'=3↑⎕STATUS)≠3)
[  7]   ENDLOOP:  ⎕UNTIE 1
[  8]   M←((I-1),79)⍴,M
[  9]   ⎕TC[5]
[ 10]   0 0⍴257 ⎕SYS 45191
[ 11]   M
```

Several publications available from APL Press, Suite 201, 220 California Ave., Palo Alto, CA 94306 are of interest to teachers. Some titles: Introducing APL to Teachers, APL in Exposition, A Source Book in APL--Papers by Adin D. Falkoff and Kenneth E Iverson, APL and Insight, and Starmap. Information and prices on these and other publications from APL Press are available from them on request.

ON POSITION-INDEPENDENT CODE AND LOADING ML ROUTINES IN LANGUAGE

The 6809 microprocessor was designed to allow programmers to write and use code which can be located anywhere in memory (position-independent code, or PIC). Such code need not be linked (tied to a specific set of addresses) and runs right well wherever you put it. After you load such code, you need to know only its starting address to execute it.

Not an assembly-language programmer and not interested? You need know nothing about assembly language to use PIC code in high-level languages. You can 1) copy it (thine eyeballs will ache), or 2) copy the assembly language program and assemble it. Anybody who can breathe and type can do it. (See Vol I, p. 142 ff.) SuperPET does all the work and makes a file you can use directly to load ML code into the languages.

Why should you bother? What are PIC's advantages? (1) Programs are modular: Any PIC module may be used in any program in any language we've tried so far. You can use as many PIC modules as memory can hold in any specific program. (2) Programs are independent: Every PIC module is self-sufficient; it works wherever the language may locate it. (3) Programs are integrated: You need never set MemEnd_ or go through fancy procedures to use PIC modules. They are loaded from, and are a part of, a program in language. You'd never know you were using ML routines--except for the speed of execution. In sum, you can write or use PIC modules in a high level language to do swiftly what the language itself may do very slowly or not at all; you may use that same PIC module anywhere.

We received a classic example of how to write and use PIC from Associate Editor Stan Brockman. Because it is a very short machine-language dump, anybody with a printer can use it to find out how to write and use PIC. The rest of this arti-

cle is in three parts: 1) How PIC works, 2) How to write it, and 3) How to use it in mFORTRAN, in a very direct and simple way. Later articles this issue will show more advanced ways to use PIC in the languages.

1 - HOW PIC WORKS Every 6809 ML routine is executed step by step, with the Program Counter (PC) telling the 6809 the address of the next instruction. The 6809 dutifully gets the instruction at that address and executes it. In all machine language but PIC, the instruction addresses are absolute. The start of a routine may, for example, be given as $702C, and every instruction after that likewise is at a specific 16-bit address. If you move the code up or down in memory by a single byte, it will not work. All ML code but PIC must be linked to a specific set of addresses; that's what the Linker is for.

In contrast, PIC code is position independent. All addresses within the ML code itself are defined relative to the Program Counter, as so many bytes ahead of or behind the address which appears in the PC at a particular time. Let's look at a simple example:

Relative address in Program Counter:	Instruction Number:	Comments:
0000 [The starting address,	1	The first instruction is followed by an
0001 offset by the number of		operand of two bytes (we don't care what).
0002 bytes at the left.]		
0003	2	The second instruction, followed by another
0004		operand of one byte.

Isn't it obvious that instruction 2 is always in the Program Counter when the PC holds the starting address plus 3 bytes? If the starting address is $6000, instruction 2 starts at $6003; if we start at $7000, we find instruction 2 at $7003. Simple arithmetic will keep track of the address of any part of our code; in fact, the the assembler itself will tell the 6809 to find the address of any piece of code if we use PC relative addressing (PCR). We must tell the 6809 only one thing when we execute PIC code: the address at which the code starts.

2 - HOW TO WRITE PIC CODE: Let's take a look at Stan's program to see how he uses PCR to generate his PIC code. We call this package pcr.asm:

```
openf_    equ $b0ae          ;We assign the ROM addresses of system routines
closef_   equ $b0b1          ;so we do not have to link this code after it is
fputchar_ equ $b0cf          ;assembled. Linking merely assigns addresses.

          ldd   $0122        ;Stan revised some old PIRQ code for this demo.
          addd  #$7fb0       ;The code itself is not important, but the use of
          std   istop,pcr    ;PCR is--as here, where a label is offset, PCR.
          leax  mode,pcr     ;And another label here. Each label and each sub-
          pshs  x            ;routine call is PCR offset--except for the system
          leax  file,pcr     ;routines in ROM whose addresses we define in the
          tfr   x,d          ;equates at the very start of this program.
          jsr   openf_       ;WARNING! Do not attempt to assemble this code in
          leas  2,s          ;the V1.0 assembler, which will refuse it. Use the
          std   outp,pcr     ;V1.1 assembler only!
          if    ne
            ldd   #$8000     ;There is no advantage to PCR addressing when pro-
```

```
            std   line,pcr      ;grams are to be linked for a specific starting
            loop                ;address and will always be used at that address.
              ldx  line,pcr     ;The code runs more slowly than it otherwise would
              jsr  print,pcr    ;because of the added arithmetic. In addition, no
              ldx  line,pcr     ;starting address is included in the code (since
              ldb  #80          ;it is not linked); it cannot be loaded either
              abx               ;from menu or in the monitor unless it is linked.
              stx  line,pcr     ;Nothing keeps you from linking PIC such as this
              cmpx istop,pcr    ;for use at a specific address (it works fine),
            until ge            ;but why go to the trouble of writing PIC if it
            ldd outp,pcr        ;is to be used at a fixed address? On UNIX and
            jsr closef_         ;UNIX-like systems (such as OS-9), however, PIC
          endif                 ;is invaluable, for it can be put anywhere in mem-
          rts                   ;ory and will run there without problems. Stan's
                                ;use of it in mFORTRAN and our later use in mBAS-
;subroutine and data follow:     SIC show other applications of PIC.
print     tfr   x,d             ;Remember that this routine, once assembled, need
          addd  #80             ;not be linked (which means assigned to specific
          tfr   d,y             ;addresses). The assembled code in the .b09 file
          loop                  ;may be put directly into memory and executed by
            ldb  ,-y            ;a call to the starting address. There are several
            cmpb #32            ;ways to do this; Stan simply gets the .b09 file
            quif ne             ;and stuffs it into an mFORTRAN program as a char-
            cmpy line,pcr       ;acter variable, as we'll later show.
          until eq
          tfr   y,d             ;If you prefer, you can get the code from the LIST
          subd  line,pcr        ;or .lst file, after assembly, and transcribe it
          addd  #1              ;by hand (if fond of copying). It's far less work
          tfr   d,y             ;to pull the .b09 file into your language code and
          loop                  ;use it without transcribing (though you'll have
            ldb  ,x+            ;edit a bit before you use it.)
            pshs y,x,d
            ldd  outp,pcr       ;Stan thoughtfully put the output file designation
            jsr  fputchar_      ;at the very end of this code. If your printer is
            leas 2,s            ;"printer" and not "ieee4", it's simple to revise
            puls y,x            ;the ASCII code (in hex) at the end of program to
            leay -1,y           ;say "printer" instead, without reassembly. The
          until eq              ;program in language is commented to show where
          ldb  #$0d             ;the code is, so you can change it--to serial, if
          pshs d                ;you wish.
          ldd outp,pcr          ;If you'd rather use this routine to disk, change
          jsr  fputchar_        ;label "file", below, as shown at end of program,
          puls d                ;and then assemble it.
          rts
istop     rmb 2                 ;Each of these labels is always referred to PCR,
outp      rmb 2                 ;wherever employed in the code above.
line      rmb 2
mode      fcb 'w,0
file      fcc "ieee4"           ;Change this to "printer" or "serial", or to
          fcb 0                 ;"disk.filename", as you wish.
          end
```

Note that all system addresses are handled as absolute values; all labels are specified PCR. A note on stack handling is found at the end of this article.

3 - A SIMPLE EXAMPLE OF HOW TO USE PIC CODE IN MFORTRAN: The code generated by the assembler is called the pcr.b09 file. For this demonstration, we bring that file directly into the microEDITOR (used alone or in Development--never in a language; you'll ruin the file!). We must edit it, for it holds some English phrases we must get rid of, and we must assign the code to a character variable. How do we stuff "a='fc0122...'" on the line below (the first line of code) to assign it to character variable "a" without running the end of the 80-character code line into oblivion? That's easy. We duplicate each line in the mED--and we then erase the last half of the original line, and the first half of the duplicate line, like this:

```
fc0122c37fb0ed8d007f308d00813410308d007  [last half of line erased]
  [first half of duplicate erased]     d1f10bdb0ae3262ed8d006c2727cc8000ed8d0065
```

Now we can assign each line to a character variable by concatenating:

```
a="fc0122c37fb0ed8d007f308d00813410308d007"
a=a//"d1f10bdb0ae3262ed8d006c2727cc8000ed8d0065"
```

We repeat the process for all lines and file all lines of the code to disk, say as the file "pcr.char". We can now get the code off disk into mFORTRAN so we can use it. Be sure to have the quotes shown above around the code when filed.

In language, we face a second problem. The code, as a character variable, is not executable. In memory, the first character ("f") will show as $66. We don't want $66; when we look in memory, we want to see the exact code printed above. In SuperPET, there's a system routine to do this--HSTOB_ (Hex String to Binary), at address $B01E. So Stan SYS's that routine to convert the code from its character representation to executable binary code. If you look in the monitor after character "a" is converted, you'll see an exact duplicate of the code above.

The last problem is to find the address of the character variable after it has been converted to executable code. In mFORTRAN, intrinsic function VARPTR will find the the starting address for us. If we now SYS that address, the code will execute. Remember that as an mFORTRAN program runs, the addresses of character variables may change, so we must find the address of our PIC code each time we use it by a call to subroutine "mldump".

```
subroutine mldump
character a

a="fc0122c37fb0ed8d007f308d00813410308d007"       * The code to the left was
a=a//"d1f10bdb0ae3262ed8d006c2727cc8000ed8d0065" * formed in the mED, loaded
a=a//"ae8d0061ad8d0019ae8d0059c6503aaf8d0052a"    * alone, exactly as we descri-
a=a//"c8d004a2de7ec8d0046bdb0b1391f10c300501f02" * be the process in the text
a=a//"e6a2c120260710ac8d003226f31f20a38d002ac"    * above. We haul this file in-
a=a//"300011f02e6803436ec8d001bbdb0cf3262353031" * to mFORTRAN and write this
a=a//"3f26edc60d3406ec8d0008bdb0cf35063939ff0"    * program to use it.
a=a//"0ff770077006965656534 00"//char(0)
*            w   i e e e 4  Change this hex code for your printer filename.
*           'w' stand for 'write'
ia=varptr(a)                 * Integer "ia" points, through VARPTR, to string a.
j=sys(cnvh2i('b01e'),ia,ia)* We SYS a routine to convert a hex string to binary
                             * (hstob_, p. 173, Assembler Manual, at $B01E).
```

```
j=sys(ia)                    * And SYS to the address of the ML program.
a=""                         * Be SURE to null "a" or you'll run out of memory
end                          * if you use the dump repeatedly.
```

* * *

Stan Brockman warns those who write PIC code to beware on stack handling. When you SYS, the return address for mFORTRAN is pushed on the stack; if you pass two or more parameters, they are also stacked (only parm 1 is passed in the D accumulator). If, in your PIC routine, you manipulate the parms by a PULL or PUSH, the Stack Pointer will be offset so that you will not return from the SYS to language--but will crash. In place of PUSHes or PULLs, use offset addressing (as at left)--and leave the Stack Pointer alone. We ran a little SYS program in mFORTRAN and recorded the D register, the Stack itself, and the Stack Pointer after passing three parameters. Parameter one was in the D register, as expected. Our code started at $7500, so we SYS'd our routine as follows: i=sys(cnvh2i('7500'),01,02,03). After we passed the three parms 01, 02, and 03, the stack and stack pointer were as follows:

```
RIGHT          WRONG          Action:
std P1,pcr     std  P1,pcr    ;store Parm 1
ldd 2,s        puls d         ;get Parm 2
std P2,pcr     std  P2,pcr    ;store Parm 2
```

Stack location relative to Pointer:	Stack Contents:
+4 bytes	00 03 (Parameter 3)
+2 bytes	00 02 (Parameter 2)
Stack Pointer--> 0	9A B6 (Return Address to mFORTRAN)

In short, when you SYS, the Stack Pointer points to the return address to the language. If you move the pointer, 1) at least move it back there, and 2) don't adjust the stack pointer to clear out the parms. The SYS routine does that for you. The programmer normally must clean up the stack when passing parms. Don't!

VARIABLE FILES AND PIC CODE A VARiable file is most useful for saving machine language code, because you can write to and read from VAR files any values, including the CONTROLs (ASCII 0 through 31). All machine language code is full of such values. When you read a file designated as a VAR file, CONTROLs are read as data values, and not as control values. A carriage return, for example, is read as decimal 13, not as end-of-line or record; ASCII code 12 is read as such, and not as a command to clear screen and home the cursor. All .mod files created by assembly and linking in SuperPET are, for this reason, filed to disk as VAR files in PRG format.

What is a VAR file? It can take any number of forms; either 1) the pre-defined format expected by SuperPET's operating system, or 2) any format you create to work in a certain way. We'll say more on VAR files in general next issue. Meanwhile, let's define the type we'll use to hold PIC code.

We'll create a VAR,USR file which is prefaced by two data bytes which define the length of the executable code in the file. If the executable code is $80 bytes long and the first byte of executable code is $FC, then we want the first three bytes in the file to be those at the left. Both mBASIC and mFORTRAN will load a file designated as a VAR file by throwing away the first two bytes--the length bytes--and then starting the load with $FC. The

```
00 80 FC
```

VAR files we make can be read in mFOR and mBASIC and probably in all languages.

The statement in the mBASIC manual that it doesn't support VAR filetype is only true in part. mBASIC can't form VAR files as such, but it can be tricked into making them--we simply write a text file and read it as a VAR file--anywhere. There is one minor flaw in mBASIC (but not, so far as we know, in any other language); mBASIC strips all CR's from a VAR file as it's brought into memory. It is easy, however, to replace the CR's during loading if you know about this.

LOADING PIC INTO microBASIC AND OTHER LANGUAGES FROM DISK

Can't we find a simple way to convert a PIC .b09 file into a disk file which we can load into any language whenever we need it? Well, with a great deal of help from Associate Editors Stan Brockman and Terry Peterson, we indeed can. So far, we can do it in both mBASIC and mFORTRAN--using the same disk file for both languages. We suspect that disk file will work in all of the languages if the proper loading program is written in each language. Credit Terry Peterson for insisting that VAR (variable) files were the proper file format for PIC code in the languages, and for chrome-plating our programs.

Because we want a file of machine code, we use VAR files in USR format (it's easy to remember what they're for). The first two bytes must specify the length of the executable code in the file. "pcr.b09", the file we convert, says that its code is $96 (decimal 150) bytes long--but as with all .b09 files, Waterloo either started counting bytes with zero or plain goofed. All .b09 files are one byte longer than they say they are; we have to add that to reported file length. (The two bytes of length data with which we start the file are not counted.)

To make what we are about to do clear, let's look at the .b09 file itself. Note that it is prefaced by data and English phrases. These we must get rid of. We'll also add one to the file length shown. Note the carriage return in the last line, marked with two carats. We'll have to patch in every CR when we read the file in mBASIC.

```
Errors 0
$0096 ; Length
Object
fc0122c37fb0ed8d007f308d00813410308d007d1f10bdb0ae3262ed8d006c2727cc8000ed8d0065
...several lines removed
3f26edc60d3406ec8d0008bdb0cf35063939ff00ff77007700696565653400
       ^^
```

TO CREATE AN EXECUTABLE DISK FILE: This first step we take only once. Thereafter, the disk file of executable code may be used as often as we need it. Why must we convert "pcr.b09" to executable code? Isn't it executable already? Well, we're going to load it into language as a string value; the first characters in the .b09 file are hex "fc", which will form $66 ("f"), $63 ("c"), etc. Instead, we want hex "fc" to translate as one byte to itself (decimal 252). The program below forms a disk VAR file which loads to form an exact duplicate of .b09 file.

```
110 ! con_to_var:bd. Loads and converts the .b09 PIC file to a (VAR) file.
120
130 print chr$(12); : open #12, "pcr.b09", input
140 open #14, "(t)pic_code,usr", output          ! Form as TEXT file, read
150                                             ! as VAR file.
160 on eof ignore
170 input #12, a$,b$                  ! Read ERRORS (throw away), then $LENGTH.
180 b$=hex$(hex(b$(2:5))+1)           ! Add one and convert length to hex.
190 print #14, chr$(hex(b$(1:2)));chr$(hex(b$(3:4)));  ! Print LENGTH to file.
```

```
200 loop
210   linput #12, trash$              ! Get rid of remaining English words.
220 until trash$="Object"
230 loop
240   linput #12, a$                  ! Get object code.
250   if io_status then quit
260   for i%=1 to len(a$) step 2      ! Parse code 2 bytes at a time and
270     print #14, chr$(hex(a$(i%:i%+1)));   ! convert it for disk file.
280   next i%
290 endloop
300 print #14 : reset                 ! End file with a CR!!!
310 call read_var : stop
```

We include with the program the short procedure below, which will read the VAR file so created to the screen in hex. If you vary the conversion program, you'll need it--unless you'd rather spend your time block-reading your disk.

```
350 proc read_var                                ! Reads all VAR,USR and VAR,PRG
360 print chr$(12);                              ! files to screen in hex.
370 endd$=chr$(0)+chr$(13)+chr$(13)              ! This definition of EOF seems
380 open #20, "(t)pic_code,usr", output          ! to catch the end of all PRG and
390 on eof ignore                                ! VAR files you wish to read.
400 loop
410   get #20, a                                 ! Also reads all APL VAR files,
420   a$=a$+chr$(a)                              ! whether SEQ or PRG.
430   if idx(a$,endd$) then quit
440   code$=hex$(a) : print code$(3:4);" ";
450 endloop
460 endproc
```

LOADING A VAR FILE INTO LANGUAGE If you're not an mBASIC programmer, skip this section and proceed to the article on loading PIC from disk in mFORTRAN. In the program below, we load PIC code into mBASIC. Once it's loaded, you may call it at any time so long as the string which holds it (pic$) is not nulled. The code lines below must be the very first of your program, so that pic$ will be the first string identified in the variable table and easy to find.

```
100 ! picload:bu. Load PIC code from (var) file on disk, ready to use.
110
120 on eof ignore
130 open #12, "(v)pic_code,usr", input   ! Read as a (VAR) file; SPET looks
140 loop                                ! to the first two bytes for length
150   linput #12, a$                    ! of record on all (VAR) files.
160   if io_status then quit
170   pic$=pic$+a$+chr$(13)             ! Replace CR's removed during disk load
180 endloop
190 loaded=1 : close #12                ! End of load.
195
200 ! Insert any amount of your own program here.
203
205 call find_do("3511210599")    ! Call PIC. Mad search string explained below.
```

The PIC module is short; the load from disk swift. If you now search high memory you'll find an exact image of the .b09 file. Mark the address; SYS it; the code

executes. But how do we avoid a long, slow search in the monitor for the address of the module? And will the address not change as we null or revise strings? It will. We found a short, swift, simple way to locate and SYS any PIC code:

```
210 proc find_do(search$)                              ! Pass the search parm.
220   address%=0 : var$=""
230   start_var%=peek(hex('44'))*256+peek(hex('45')) ! Find start of var. table.
240   end_var%=peek(hex('46'))*256+peek(hex('47'))   ! Find end of var. table.
250   for i%=start_var% to end_var%
260     var$=var$+value$(peek(i%))                   ! Search table for pic$
270     if idx(var$, search$)
280       address%=peek(i%+1)*256+peek(i%+2)         ! When found, get address.
290     endif
300     if address% then quit
310   next i%
320   if address% and loaded then sys address%       ! And SYS that address.
330 endproc
```

An Explanation of Procedure Find_Do: Though the procedure above is fast and simple, and we can't measure the delay while it finds and executes PIC code, it requires some explanation. Simmer down, Bodsworth; if you don't want to know why it works, skip the details which follow.

Thanks to Gary Ratliff and Loch Rose, we know about two pointers in mBASIC which make the job easy. The first pointer, at $44-$45, points to the start of the Table of Single Variables; the second, at $46-$47, shows the end of the table. If we know what to look for, we can locate the address of "pic$" in the table. How?

From Gary Ratliff's work, we know string values are identified in the Variable Table with a hex number in front of the string name. The three high bits of the number are always 001 for a simple string, as at the left. The low five bits say how long the string name is. We know that "pic" is a three-character name (00011, as at left), so we know "pic" will be identified with a hex number which, in binary, is 0010 0011. In hex, that's 2 (high nybble) and 3 (low nybble). We look for $23. Because all PEEKs come back in decimal, we first search for the decimal equivalent--35; when we find a 35, we know we're about to read a string name. Is it "pic$?"

```
High 3 Bits:  Low Five Bits:
0   0   1     0  0  0  1  1
```

Our PEEKs will come back in decimal ASCII codes; "p" is 112, "i" is 105, "c" is 99. So we must scan the variable table for this sequence: 35 112 105 99. Because string numbers aren't separated by spaces, the true sequence is "3511210599"--which explains the mad IDX search string in the loader program above, and also explains how to create your own search string for any PIC string name--except for even-length string names. Hold tight, Bodsworth. What about 'em?

In the Variable Table, Waterloo converts all even-length strings to odd-length by adding a space (ASCII 32) right after the string identifying number. Suppose "picc" is a four-character string name. It's made five characters long, so we parse the identifier at left, to read $25 or decimal 37. We therefore search for "37 32 112 105 99 99", without spaces, of course. The added space is underlined in the line above.

```
0 0 1  0 0 1 0 1
string   len=5
```

We've now located both identifier and name of "pic$" in the Table. Where's its address? In the next two bytes. We read the address in line 280 of "find_do"; we

SYS it. That's it. Remember to use find_do each call; the address may change. Since search strings are passed to find_do as parms, you can use the procedure with any number of PIC modules in memory.

When you write and amend a program using PIC, you'll bury pic$ in obsolete trash in the variable table as you revise. File program to disk and reload to get pic$ up front again. When the program is run as loaded, pic$ stays up front.

A warning: If you switch from mBASIC to its microEDITOR, all variables go null, including pic$. If you return to mBASIC and call find_do, guess what happens... We therefore set variable "loaded" to 1 (see program "picload-bu") when we load pic$. Note find_do won't work if you switch to mED and back--loaded becomes 0. It's a safety precaution--use it!

OTHER WAYS TO HANDLE PIC CODE IN THE LANGUAGES

It's obvious that if we can load PIC code from disk in mBASIC, we can do it even more simply in the other languages; we needn't fuss around with string identifiers or with procedures to find the code. Intrinsic functions analogous to VARPTR in mFORTRAN will easily find the address of PIC code for us.

We print below two mFORTRAN programs, either of which will load and execute PIC code stored on VAR files. They use the disk file, "(v)pic_code,usr" which we created in the mBASIC program "con_to_var:bd" in previous pages, this issue. The file is $97 or decimal 151 bytes long.

You can load the code into mFORTRAN at any time, and call it at any later time. The programs below, to save space, load and execute the code in one swoop.

```
*program stanload:fd   By Stan Brockman.
  character pic,a
  integer ctr
  open(9,file="(v:151)pic_code1,usr")
  pic=rpt(' ',151)
  ctr=0
  while(1) do
    read(9,*,end=1) a
    il=len(a)
    pic(ctr+1:il+ctr)=a
    ctr=ctr+il
  endwhile
1 ii=varptr(pic)
  print,ii     * This line and next for test only. Remove both PRINT and PAUSE
  pause        * statements for full execution, since PIC address may change!
  i=sys(ii)
  a=''         * Be tidy, Bodsworth. Null a and save some memory.
end
```

This program, to line 1, loads PIC code from a VAR file. The portion from line 1 onward executes it. If you want to call PIC by subroutine, write one with these two lines:

```
ii=varptr(pic)
i=sys(ii)
```

Insert nothing between the call to VARPTR and the SYS, for the address of PIC can change!

```
*program loadvar:fd
  character pic
  pic=rpt(" ",152)
  open(9, file="(v:151)pic_code1,usr")
  read(9,*) pic(1:151)
  close(9)
  ipic=varptr(pic)
  i=sys(ipic)
end
```

```
* The length of PIC must be defined as
* one byte longer than the true VAR file
* length in this version. We don't know
* why, but the program won't work unless
* it's done.
```

As you can see, it's simple to load PIC from a VAR file, and easy to execute it with intrinsic function VARPTR.

Are there any other ways to store PIC code and use it in the languages? Yes. We have successfully loaded PIC code from plain TEXT,SEQ files, but the presence of carriage returns in such files makes them hard to load in mFORTRAN and, we suspect, in most languages. The VAR,USR file loads CR's as well as anything else.

Stan Brockman favors VAR,PRG files. He simply links the PIC code and then loads the .mod file created by the linker. But there are a few contortions. He must link them to an origin four bytes higher than the .b09 file length, and read the file in mFORTRAN with a record length from two to eight bytes longer than .mod file length. Last, he has to SYS an address several bytes higher than the start of the loaded code. The first six bytes in every .mod file contain loading information and make loading a problem.

Stan managed to work around them with a program which 1) reads the file in a preliminary pass, 2) loads it after the true length is known, and 3) then SYS's the start-of-code. We figure VAR,USR files are easier to understand and use. Nobody has yet tried to load a block of PIC code longer than 512 bytes from any file of whatever kind. Be interestin' to see what happens!

WE SORT OUT SOME SORTS We previously asked readers to send in some neatly structured string sorts; we not only received a large bunch, but gained much insight into the proper way to employ various types. It's clear that there's no such thing as a "best" sort. Before we go into the why of that conclusion, we note that the speed of a sort depends on how you format it. Use integer variables (not reals) and multiple statements per line (not singles) and your sort will execute 30% faster in mBASIC. In the other languages, use integer variables, not reals. The difference in speed is substantial. Note also that the sorts printed below should be easy to convert to other languages.

Well, why do we say there's no such thing as a "best" sort? In I, 270, we printed a modified shell sort. This issue, in the pages following, you'll find two more: 1) a structured version of Hoare's Quicksort, sent in by Jerry W. Carroll, and, 2) a double-bubble sort from Robert Dray. Your first reaction, we guess, is that nobody in his right mind ever uses a slow bubble sort. Dead wrong. Compare performance of the double-bubble, shell, and Hoare sorts on randomly arranged (disordered) lists and on almost-ordered lists below. The "almost-ordered" list was in alphabetical order except for six names added at the top of their alphabetical place (e.g., "Axord" at the top of the A's; "Blucher" at the top of the B's). This list is the "ordered" one below. Data in seconds to execute the sort:

Array Size	Quicksort		Double Bubble		Shell Sort	
Sorted:	List Random	List Ordered	List Random	List Ordered	List Random	List Ordered
25	4	5	14	2	7	3
75	14	33	98	6	28	12
150	33	147	407	10	75	28
300	79	749	...	16	256	55
600	205	...	...	..	...	..

Hoare's Quicksort obviously is best at sorting random lists, but is miserable at handling a list which is almost sorted. The double bubble likewise is miserable

when it sorts a random list but performs well on ordered ones. Our poor shell can't beat the other two at their best, but it wins over the bubble at sorting random lists and whips Quicksort on ordered lists. Because of this, we designed ALPHA (the assembly language sort on ISPUG's utility disk) as a shell sort.

Both sorts following are set up with input sections so you avoid a few problems in DIMensioning and the proper Option Base. The first is Robert Dray's Double-Bubble. He scans the list alternately from top to bottom and bottom to top, so that "heavy" items bubble down and "light" items bubble up. But don't add new items to an already-sorted list either at the very top or at the very bottom--if you do, comparative time to sort rises by a factor of 10. Take the time to put items in their alphabetic niche ("Axord" at the head of the A's, as we did), and the double-bubble will do a swifter job for you than any other sort if there are relatively few changes to an already-sorted list.

```
100 ! double_bubble:bd. From Robert Dray. Use OPTION BASE 1.
110 print chr$(12); : option base 1
120 input "Enter filename of file to be sorted: ", file$
130 print : input "Enter number of items to be sorted: ", num_items%
140 dim list$(num_items%) : open #12, file$, input
150 on eof ignore
160 for i%=1 to num_items%              ! The list on disk must be formed
170   linput #12, list$(i%)             ! with a single entry on each line.
180   if io_status then quit
190 next i%
200 reset : print "Sorting ..." : t1=time : call double_bubble
210 t2=time
220 mat print list$
230 print "Time to sort:";t2-t1;"seconds"
240 stop
250
260 proc double_bubble
270 upper_bound%=num_items% : lower_bound%=2
280 last_swap_up%=upper_bound% : last_swap_down%=lower_bound%
290 loop
300   for j%=upper_bound% to lower_bound% step -1
310     if list$(j%-1) > list$(j%)
320       exchange$=list$(j%) : list$(j%)=list$(j%-1)
330       list$(j%-1)=exchange$ : last_swap_down%=j%
340     endif
350   next j%
360   if lower_bound%=last_swap_down%+1 then quit
370   lower_bound%=last_swap_down%+1
380   for k%=lower_bound%-1 to upper_bound%
390     if list$(k%-1) > list$(k%)
400       exchange$=list$(k%) : list$(k%)=list$(k%-1)
410       list$(k%-1)=exchange$ : last_swap_up%=k%
420     endif
430   next k%
440   upper_bound%=last_swap_up%
450 endloop
460 endproc
```

Jerry W. Carroll obviously spent a lot of time pulling Hoare's Quicksort out of the maze of GOSUBs and GOTOs in which it was buried. Don't protest because he

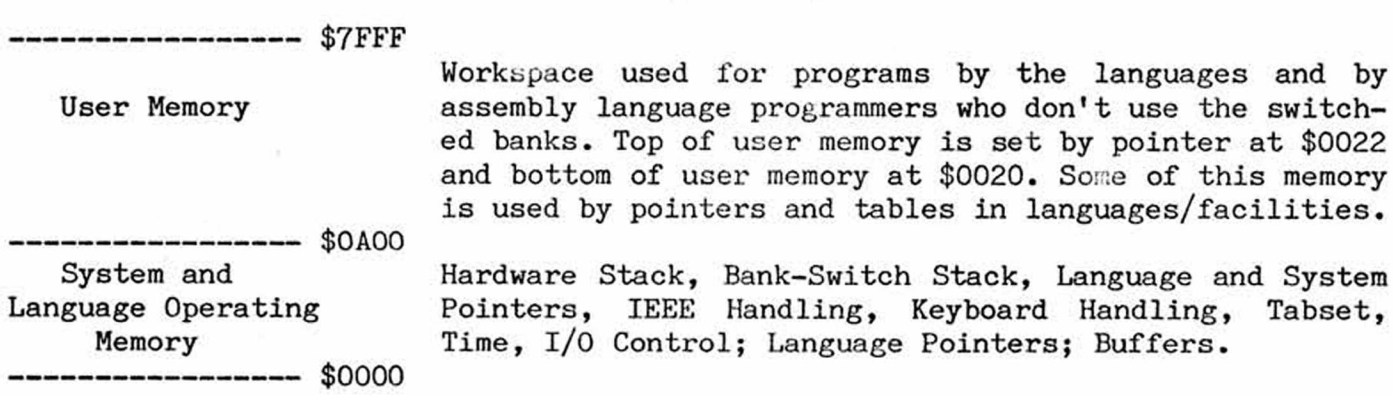
uses some single-character variable names in the sort. If the names were 31 characters long, we wouldn't know any more about how the sort works than we now do.

```
100 ! "shortsort:bd". A Quicksort routine from Jerry W. Carroll.
110
120 print chr$(12); : one%=1
130 input "Enter no. of items to sort: ", n%
140 print : input "Enter filename of list to be sorted: ", file$
150 dim string$(n%+one%) : open #4, file$, input
160 for i% = one% to n%
170   input #4, string$(i%)
180 next i%
190 reset : t1=time                    ! Reset closes files in V1.1
200 if n% <= 9
210   call straight_insertion_sort
220 else
230   call quicksort                   ! The MAT PRINT statement (line 260) will
240 endif                              ! print chr$(0) and chr$(255) to screen
250 print "Sorting is done ..."        ! at end of sort. We used MAT PRINT to
260 mat print string$                                  ! save space. A loop
270 print "Time to complete sort was:";t2-t1;"seconds." ! will get them out.
280 stop
290
300 proc quicksort
310 ! Quicksort pp. 116-117, Donald Knuth, vol. III - Sorting & Searching
320 ! string$(n% + 1) is array of strings to be sorted.
330 ! n% = number of items in array excluding string$(0) and string$(n%+1)
340 ! m% = max. no. of items in largest partition for straight insertion sort.
350  print "Sorting has started ..."
360  string$(0)=chr$(0)                ! lower boundary of array
370  string$(n%+one%)=chr$(255)        ! upper boundary of array
380  m%=9 : p%=one% : r%=n%            ! stage Q1
390  loop                              ! start of new stage Q2
400    loop
410      done%=one% : i%=p% : j%=r%+one%  ! Boolean argument
420      key$=string$(p%)
430      loop                          ! stage Q3
440        i%=i%+one%
450        while string$(i%) < key$ and i% <= j%
460           i%=i%+one%
470        endloop
480        j%=j%-one%                  ! stage Q4
490        while key$ < string$(j%) and j% >= i%-one%
500          j%=j%-one%
510        endloop
520        if j% <= i%                 ! stage Q5
530          hold$=string$(p%) : string$(p%)=string$(j%) : string$(j%)=hold$
540        else                        ! stage Q6
550          hold$=string$(i%) : string$(i%)=string$(j%) : string$(j%)=hold$
560        endif
570      until j% <= i%
580      if r%-j% >= j%-p% and j%-p% > m%   ! stage Q7
590        s%=s%+one% : stack%(s%,one%)=j%+one% : stack%(s%,2)=r% : r%=j%-one%
600        r%=j%-one% : done%=0
```

```
610       elseif j%-p% > r%-j% and r%-j% > m%
620         s%=s%+one% : stack%(s%,one%)=p% : stack%(s%,2)=j%-one%
630         p%=j%+one% : done%=0
640       elseif r%-j% > m% and m% >= j%-p%
650         p%=j%+one% : done%=0
660       elseif j%-p% > m% and m% >= r%-j%
670         r%=j%-one% : done%=0
680       endif
690     until done%
700     if s% > 0
710       p%=stack%(s%,one%) : r%=stack%(s%,2) : s%=s%-one% : done%=0
720     endif
730   until done%
740   call straight_insertion_sort
750 endproc
760
770 proc straight_insertion_sort
780   for j%=2 to n%
790     i%=j%-one% : key$=string$(j%)
800     while key$ < string$(i%)
810       string$(i%+one%)=string$(i%) : i%=i%-one%
820     until i%=0
830     string$(i%+one%)=key$
840   next j%
850   t2=time
860 endproc
```

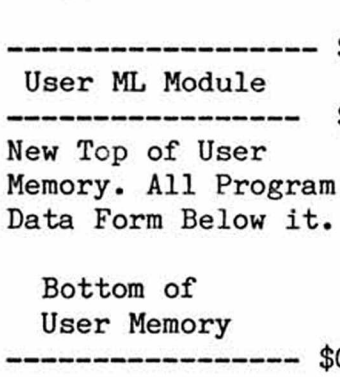
A MAP OF THE BATTLEFIELD A charter member of ISPUG, an applications programmer, scarred and weary, dropped us a note: "While I don't want to become a machine language programmer, I'm often driven in that direction by specific problems which I can't solve in high-level language--but I don't know enough to start. The Development manual leaves me gasping; your authors say, for example, to load ML routines in the "top of user memory;" why? I think I know, but really don't; I don't understand linking, locating, getting rid of a module when you no longer need it. In short, I need a map of the fundamentals." So we take up silicon cartography and start with SPET's address space, for our old hand is not the only confused reader (we exclude poor Bodsworth, who is always confused):

System Addresses	Comments:
---------- $FFFF	
ROM Routines from $A000-$FFFF	System Library and System Routines--24K bytes. ROMs are often identified as A block, B block, C block, etc. Selected ROM routines are found, by address, on disk as "watlib.exp" (Waterloo Library Exports). Floating point (decimal arithmetic) routines are filed as "fpplib.exp"
---------- $9FFF	
The Switched Banks	One of 16 switched banks of 4K bytes. Current bank is shown at $0220 as $0 through $F. All languages interpreters, linker, assembler load here. 64K bytes.
---------- $8FFF	
Hyde	Memory above screen is used in part by Waterloo.
---------- $87D0	
Screen Memory	Decimal 2000 bytes allocated to screen memory.



If you add up the memory space above, with one switched bank in operation, it totals 65,536 bytes, which is the maximum address space of the 6809.

Let us now put a microscope on User Memory and answer the questions about using that space. Below, we show a small ML module at $7F00, with MemEnd_ set also at $7F00. It's a convention that assembly language programs be located at the very top, just below $7FFF. There are two good reasons to use the top, not the bottom at $0A00. First, if you goof, and your ML module is larger than you thought, the tail-end will appear on screen as a warning--the overflow visibly prints. Second, if the ML module were placed just above $0A00 and then overflowed, it'd overwrite whatever program you used, and you'd crash.



When you load an ML module in high user memory, you set the pointer for MemEnd_, at $0022, to the exact starting address of that module. Whatever language or facility you then load is forbidden, by that pointer, to invade the memory space above the pointer. If you don't reset MemEnd_, the language or facility you load will, sooner or later, overwrite your ML module. Examples: 1) strings are stored just below the location defined by MemEnd_. If you put an ML module at top of user memory, the strings almost always overwrite it; 2) the languages/facilities set up some pointers/tables very near MemEnd_. They also overwrite ML code.

All the languages and facilities in SuperPET obey the MemEnd_ pointer at $0022--if you load the language from menu after you reset MemEnd_. Once MemEnd_ is set, it remains at that value until you change it, turn off SuperPET, or switch to 6502 mode. In all languages but mBASIC, do not reset MemEnd_ while a program is loaded, for all its internal pointers are adjusted to that MemEnd_ value. In mBASIC only may you switch to mED, enter the monitor, change MemEnd_, and return safely to language. Calling mED resets all pointers to "just loaded" values.

MemEnd_ is set or reset in the monitor by dumping memory location $0022 (>d 22 is the command), by overtyping the first two bytes with 7f ff or whatever value you want, and by hitting <RETURN> on that line. Then >q the monitor.

There is a quick way to reset MemEnd_ in all languages. Leave the language with a "bye"; reset MemEnd_ in the monitor at menu; then use RESET (I, 114) to return to the language. You may recover the memory previously occupied by an ML module by doing this because you also reset all pointers the language uses. In short, SuperPET acts as if the language or facility were just loaded. RESET is found on the ISPUG Utility and Starter-Pak disks.

We've now drawn a general map, explained MemEnd_ and how to protect ML modules in high user memory, how to get back the space occupied by such modules, and how to set and reset SuperPET's language and facility pointers safely.

Which leaves the matter of "linking." (Wake up, Bodsworth.) The code created by an assembler shows relative addresses only (relative to the start of a program, defined as 00 00). Linking assigns the code to absolute addresses in memory, starting with the origin (abbreviated to "org" in the linker's .cmd file). This work comes in two parts a) telling the program itself where its parts are located (Jeez, what's the address of the first subroutine?), and b) assigning the addresses of system routines from the disk file "watlib.exp". The linker must also 1) note in the file the starting load address, 2) convert the .b09 file of unaddressed code to addressed code, 3) specify the bank in which the code is to be loaded (if it goes into a switched bank at all), and 4) state the length of the code. SPET's loader programs, either at main menu or in the monitor, read this information from the file, load the code at the addresses specified, and stop loading at the number of bytes stated in the .mod file created by the linker. If you want a simile, the assembler writes the letter; the linker addresses it; the loader programs deliver it; the 6809 reads and executes it.

UNDOCUMENTED SYSTEM ROUTINES
Part I
by John A. Toebes, VIII

We print below the first of a series of articles by John on the system routines available to the assembly language programmer in SuperPET's ROMs. John defines the routines and then shows how to use each of them. We'll continue this series until all the major undocumented routines have been covered. The Jump Table addresses shown are the actual routine addresses. If, for example, you call SPAWN_, below, at $B000, you will find a JMP to $BC75, where the routine starts. In the examples below, when John CALLs a system routine, he assumes use of CALL MACRO (I, 158) to load the D register with parameter 1 (P1) and to stack the remaining parms, if any. --Ed.

SPAWN_ : Spawn a process : at $B000; Jump Table $BC75

P1 - Address of routine to call : Result - Return code from procedure:
$0000 indicates routine successfully completed
$0001 indicates failure by the called routine.

This routine is used for calling a routine so that any of its internal routines may exit prematurely if an error condition arrises. This premature exit is done by calling the system routine SUICIDE_, which causes immediate termination, returning a failure code to the calling routine. Waterloo uses this routine in the editor to catch attempts to add a line when no more memory is available.

Upon entry, this routine preserves the contents of ExitSP_ ($002c), ExitU_ (at $0033), and CurBnk_ ($0220) on the stack. It then saves the U and S stack pointers at ExitU_ and ExitSP_, respectively. If the SPAWNed subroutine then returns through an RTS then the current bank, ExitSP_ and ExitSP_ are restored to their previous values. This method allows SPAWNed subroutines to nest correctly with a SUICIDE_, returning to the correct SPAWNing point.

Example:

```
CALL   SPAWN_, #GETMEM
IF     NE                  ;test spawn_ return code;
   JMP   NOMEM             ;if 0000, proceed; if 0001,
ENDIF                      ;print OUT OF MEMORY message.
```

SUICIDE_ : Terminate a SPAWNed subroutine : at $B003; Jump Table, $BCAB

No parameters; does not return to calling code.

This routine is only used within a subroutine that has been invoked through SPAWN_. When executed, it causes execution to resume at the point where the current procedure was SPAWNed. In addition, the routine that issued the SPAWN will be returned a result code of $0001 indicating failure. If a routine invokes SUICIDE_ and no SPAWN is active, results are unpredictable, but SuperPET usually goes into never-never land.

SPAWN_ restores the S and U registers from their saved values at ExitSP_ and at ExitU_. It then restores ExitU_, ExitSP_, and the current bank from the values saved by the call to Spawn_.

```
Example:          CALL  SPAWN_, #SUBROUT
                  ...
          SUBROUT ...
                  JSR   INTERN
                  ...
                  RTS
          INTERN  ...
                  IF    NE                ; Did we fail miserably?
                    JMP   SUICIDE_        ; Yes, go back.
                  ENDIF
                  ...
                  RTS
```

BANKSW_ : Perform a bank switched call : at $B009; Jump Table $BBF1

No parameters; does not return directly.

This routine is used to support calling routines in bank-switched RAM. It is invoked only indirectly through an ALV (Auto Load Vector) created by the linker or artificially with the assembler. Attempting to call this routine directly will meet with disastrous results.

This routine takes the next byte immediately after the call to it as the bank of the routine to be called. Immediately after the bank is the address of the routine within that bank which is called. It saves the current bank on the U stack and selects the bank of the routine to be called. It then jumps directly to the routine specified.

```
Example:          JSR   ALV29020 ; Proceed to Bank 2, call routine at $9020
                  ...
         ALV29020 JSR   BANKSW
                  FCB   $02       ;Bank to call
                  FDB   $9020     ;address of routine in bank
                  ...
```

ISHEX_ : Check for hex digit : at $B02A; Jump Table, $BA2D

P1 - Character to check : Result - TRUE ($FFFF) or FALSE ($0000) returns in D Register. CC Register ZERO Flag is Set if FALSE, or clear if TRUE; NEG flag is set if TRUE.

This routine checks to see if an ASCII character is a valid digit in a hexadecimal number. It returns a true flag for the digits '0' thru '9' and both upper and lower case 'a' thru 'f'. It does not convert the character to its binary form, but is intended to validate input before you call HEX_ or HSTOB_.

```
Example:        CALL    ISHEX_, INCHAR
                IF      EQ                  ; Is it a hex digit? (Is zero flag set?)
                    JMP     BADHEX          ; No, it wasn't.
                ENDIF
```

___RET : WSL support routine to clean the stack : at $B05A; Jump Table, $B677

X register holds unsigned number of bytes to release from the stack : No result is returned.

This routine is used by the WSL languages to clean up the stack upon exit from a procedure. It removes the number of bytes specified by the X register. If the routine is called with a JSR then the number of bytes should be incremented by 2 to account for the return address placed on the stack by the JSR. Note that execution from this routine returns to who called it, not to the calling routine.

For some strange reason, Waterloo wrote some extremely poor, slow and clumsy code for this routine, even using a loop, when the entire routine could be replaced by the 3 statements at left. Because of its design, I do not recommend that you even consider use of the routine.

```
        LEAS  X,S
        TFR   D,X
        RTS

Example:  SUBROUT LEAS  -10,S
                  ...               ; We simulate 10 bytes of used stack space
                  LDX   #10         ; with LEAS -10,S and let ___RET recover them.
                  JMP   ___RET
```

___RET2 : WSL support routine to clean the stack : at $B05D; Jump Table, $B67F

X holds unsigned number of bytes to release from the stack; no result returns.

This routine is identical to ___RET except that it also removes from the stack the first stacked parm passed the routine.

```
Example:          LDD   #10 ;Parm 2 : [Ed. Here John passes Parm 1 in the D reg-
                  PSHS  D           ; ister, and Parm 2 on the stack. ___RET2
                  LDD   #20         ; removes not only the 15 bytes called for
                  CALL  MYSUB       ; in the X register, but also Parm 2.

                  ...
              MYSUB LEAS  -15,S     ; Use up 15 bytes of storage for simulation.
                  ...               ; Then recover that space plus space which
                  LDX   #15         ; is occupied by Parm 2.
                  JMP   ___RET2
```

___NEG : WSL routine to negate an integer : at $B063; Jump Table, $B742

P1 - Signed 16 bit integer to negate : Result: Algebraic negation of P1

This routine is extremely straightforward and clean. It negates the parameter in the D register. Use it wherever or whenever useful.

```
Example:        LDD   #5
                CALL  ___NEG        ; After this, D has -5 in it.
```

___MUL : WSL support routine for multiplication : at $B060; Jump Table $B687

P1 - Multiplier : P2 - Multiplicand : Result: P1*P2, P2 removed from the stack.

This routine is used by the WSL interpreters to multiply two 16-bit signed integers, producing a 16 bit signed result. It does not check for overflow; results are truncated to 16 bits. This routine, as with all "___X" routines, clears the parms from the stack for you.

Internally, this routine is a very poor implementation of the standard shift/add algorithm for multiplication. Apparently, whoever implemented it did not understand how the MUL instruction can be extended. Note also that the primary use is for multiplication by 2 or 80. Do not use if your code is to have any resemblance of speed.

```
Example -  LDD   #80
           PSHS  D
           LDD   #5
           JSR   ___MUL  ;D reg now contains 400(5*80)
           ...   ;no LEAS is need to get rid of the 80
```

___DIV : WSL support routine for division : at $B066; Jump Table $B6CE

P1 - Divisor : P2 - Dividend : Result: P2/P1 with P2 removed from the stack

This routine is used by the WSL languages to divide one 16-bit signed integer by another. It does not check for division by zero, but there are no cases in which overflow should cause it to produce the incorrect result. The sign of the result will be negative iff both operands are of opposite signs. Removes P2 from stack.

This routine is a fairly clean implementation, although it could have been done better in places. It uses the most generic form of the shift/subtract algorithm.

```
Example - LDD   #20
          PSHS  D
          LDD   #6
          JSR   ___DIV ;D now has 3 in it
          ...   ;no LEAS is required to remove the 20
```

B I T S B Y T E S & B U G S by Gary Ratliff, Sr.

215 Pemberton Drive, Pearl, Mississippi 39208

In our last installment we introduced the notion of a cross-assembler; a cross-assembler is defined as a product which will assemble code for another processor while running in a machine which uses a different chip. Our resident SuperPET assembler uses 6809 code and has a 6809 assembler. If we designed a SuperPET assembler for the 8080, Z80 or 8088 chip, it would be a cross-assembler. Waterloo has provided a cross-assembler for the 6502 chip which opererates very much like the assembler for the 6809. The 6502 chip is only a switch-throw away for those who own the SuperPET. Those who have solved the riddle of the RESTRICTED command of my extended monitor, XMON6809, may use it to go from the 6809 side to the 6502 side and back again.

The 6502 is very similar to the 6800 in design and in the use of assembler mnemonics. When this chip was introduced, the article in BYTE was entitled, "Son of MOTOROLA." This is because some of the original designers of the 6800 decided they could do better, formed MOS Technology and developed the 6502. Because the instruction sets of the two chips are so similar, it's possible to have one .asm file do double duty in producing a version of a program which will assemble to run on either the 6809 or the 6502 side of SuperPET.

The mechanics of this idea involve the feature of conditional assembly. For those functions of the machine which require a difference in the .asm routine, we'll set a flag; when we select 1 we will assemble the 6502 version; if we set the flag to 0, we'll assemble the 6809 version.

Of course, to assemble the 6502 version of this program the user will need to have the 6502 Assembler from Waterloo. Because this product is so similar in design to the 6809 Assembler, those who are familiar with the 6809 assembler will have no trouble in using the similar 6502 product. [Ed. The 6502 Assembly Language System, including disk and manual and dongle, can be purchased from WATCOM Products, Inc. 415 Philip St., Waterloo, Ontario, Canada N2L 3X2. Last time we looked, the cost was $250 U.S. The dongle is required during assembly and linking, but is not needed when linked programs are run.]

```
; example of 6502-6809 routine assembly
; code = 1 for 6809 and 2 for 6502
    xref putchar_
    code equ 1        ; Change this FLAG
    ifeq (code - 1)
        ldx #string
    endc
    ifeq (code - 2)
        ldx # 0
        lda string,x
    endc
    loop
    ifeq (code - 1)
          ldb ,x+
          quif eq
          pshs x
    endc
          jsr putchar_
    ifeq (code - 1)
          puls x
    endc
    ifeq (code - 2)
         inx
         lda string,x
         quif eq
    endc
    endloop
    ifeq ( code - 1)
          swi
    endc
    ifeq (code - 2)
          brk
    endc
```

Here we have a simple example which we will assemble for the 6809 processor and then assemble again for the 6502. As you can see, the only change to make this program run on the 6502 is to alter the line containing the definition of code so that the value of the flag is 2.

The two different versions may be stored as dual1. and dual2.asm.

We also must create the .cmd files for the linkers to use. Again, the files are very similar. First is the .cmd file for the 6809:

```
"dual1"
org $1000
include "disk/1.watlib.exp"
"dual1.b09"
```

Here is the .cmd file for the 6502:

```
"dual2"
org $1000
include "disk/1.cbm80lib.exp"
"dual2.b02"
```

```
string fcb $0d
    ifeq (code - 1)
       fcc "hello"
    endc
    ifeq (code - 2)
        pcc "hello"
    endc
    fcb 0
    end
```

We use separate names to keep from overwriting the 6809 .map and .exp files when these are run through the 6502 linker.

Let us now assemble these two versions of the program.

```
  1                                ; example of 6502-6809 routine assembly
  2                                ; code = 1 for 6809 and 2 for 6502
  3                                    xref putchar_
  4      0001                          code equ 1          ;6809 Version

  6 0000   8E   00   11                    ldx #string
 12                                    loop
 14 0003   E6   80                           ldb ,x+
 15 0005   27   09                           quif eq
 16 0007   34   10                           pshs x
 18 0009   BD   00   00                      jsr putchar_
 20 000C   35   10                           puls x
 27 000E   20   F3                     endloop
 29 0010   3F                               swi
 34 0011   0D                       string fcb $0d
 36 0012   68   65   6C   6C              fcc "hello"
 41 0017   00                          fcb 0
 42                                    end
    0018 bytes of object code (ASM6809 V1.1  01:12:23)
```

```
  1                                ; example of 6502-6809 routine assembly
  2                                ; code = 1 for 6809 and 2 for 6502
  3                                    xref putchar_
  4      0002                          code equ 2      ;6502 Version
  9 0000   A2   00                         ldx # 0
 10 0002   BD   12   00                    lda string,x
 12                                    loop
 18 0005   20   00   00                      jsr putchar_
 23 0008   E8                               inx
 24 0009   BD   12   00                     lda string,x
 25 000C   F0   03                          quif eq
 27 000E   4C   05   00                endloop
 32 0011   00                               brk
 34 0012   0D                       string fcb $0d
 39 0013   48   45   4C   4C              pcc "hello"
 41 0018   00                          fcb 0
 42                                    end
    0019 bytes of object code (ASM6502 V1.2  01:17:30)
```

Notice that each version lists only the proper commands for the chip being used. The common portions assemble the different hex patterns required for each chip. One assembler file now serves double duty for both sides of SuperPET.

A NEW UTILITY DISK? As noted last issue, we have some gems of new programs at hand: Joe Bostic's new Editor, from which the last of the bugs are being relenlessly exterminated. Batch files which can be automatically executed are now in style; not only does BEDIT implement them, but Terry Peterson sent us a much-improved version of GSCROLL, which also implements BATCHing--as well as a selective directory capability. Using that, you can list only the files which have the names you want; if you want only files holding "aboo", you get them. Terry also sent us an improved SPMON, his extended monitor, which parses the flags in the CC register while you trace code (cheers!), and converts any ANDCC in hex or binary to show the flags affected. BATCH files we also find in John Toebes' new Command Line Monitor (CLM), an overlay on SPET's operating system, which loads from menu and gives you an arsenal of new commands and capabilities--including, again, selective directories! We don't know how or why three people invented the same capabilities at the same time, but the day of the selective directory and the batch file has arrived for SuperPET. No longer will any of us have to use separate programs to profile our printers and computers when we bootup in the mornin', nor have to read that long sphagetti list of filenames when we want a specific set of files. We hope these programs, as well as COPY/KILL (below) will be ready to issue on a second ISPUG utility disk by next issue.

Loch Rose's new COPY/KILL is not only at hand but debugged and working fine. If you hate typing filenames while you edit or copy files, you'll love this one. It lets you specify with a single keypress that a file is to be copied or deleted; better, you can juggle the order in which the files will be copied to another disk, so that the new disk will carry the files in logical order. Because Loch, like most of us, can't always remember what's in a file, he built in a command to send any SEQ file to the screen; you can abort the read at any time. This program is in machine language, loads at main menu, and is easy to use. The keys used for commands make sense; REPEAT means copy; DELETE means delete; it took us about five minutes to learn to use the delight Loch built.

Prices, back copies, Vol. I (Postpaid), $ U.S. : Vol. I, No. 1 **not** available.

No. 2: $1.25	No. 5: $1.25	No. 8: $2.50	No. 11: $3.50	No. 14: $3.75
No. 3: $1.25	No. 6: $3.75	No. 9: $2.75	No. 12: $3.50	No. 15: $3.75
No. 4: $1.25	No. 7: $2.50	No. 10:$2.50	No. 13: $3.75	Set: $36.00

----------Volume II----------

No. 1: $3.75 No. 2: $3.75

Send check to the Editor, PO Box 411, Hatteras, N.C. 27943. Add 30% to prices above for additional postage if outside North America. Make checks to ISPUG.

==========

DUES IN U.S. $$ DOLLARS U.S. $$ U.S. $$ DOLLARS U.S. $$ U.S. DOLLARS $$

APPLICATION FOR MEMBERSHIP, INTERNATIONAL SUPERPET USERS' GROUP
(A non-profit organization of SuperPET Users)

Name:_____ Disk Drive: _____ Printer:_____

Address:_____
Street, PO Box City or Town State/Province/Country Postal ID#

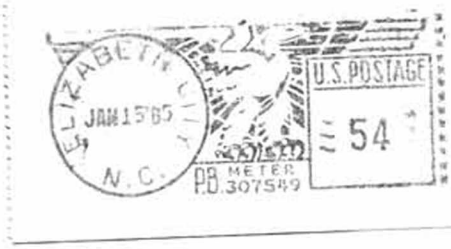
[] Check if you're renewing; clip and mail this form with address label, please.
If you send the address label or a copy, you needn't fill in the form above.

For Canada and the U.S.: Enclose Annual Dues of $15:00 (U.S.) by check payable to ISPUG in U.S. Dollars. DUES ELSEWHERE: $25 U.S. Mail to: ISPUG, PO Box 411, Hatteras, N.C. 27943, USA. **SCHOOLS!: send check with Purchase Order.** We do not voucher or send bills.

This journal is published by the **International SuperPET Users Group** (ISPUG), a non-profit association; purpose, interchange of useful data. Offices at PO Box 411, Hatteras, N.C. 27943. Please mail all inquiries, manuscripts, and applications for membership to Dick Barnes, Editor, PO Box 411, Hatteras, N.C. 27943.
 Send appropriate postpaid reply envelopes with inquiries and submissions. Canadians: enclose Canadian dimes or quarters for postage. The Gazettte comes with membership in ISPUG.

Table of Contents, Issue 2, Volume II

SuperPET Gazette
PO Box 411
Hatteras, N.C. 27943
U.S.A.



First-Class Mail
in U.S. and Canada