SUPERPET GAZETTE

Assuming you want to move up to a computer with more capability than SuperPET, what do you propose to move up to? Commodore, at the moment, offers no alternatives. Then you ask: why Commodore? We think there are five good answers when you look elsewhere: 1) unhandy to downright awful screen editors, 2) complex and very unfriendly operating systems, 3) ridiculous limits on the length of disk filenames, 4) word-processing systems rendered woefully complex by the demands of the operating system, and, 5) limits on disk drive capacity and employment. The editor started out on a CP/M machine and sold it after it took six weeks to train one secretary to use CP/M and a second quit rather than cope with it. CP/M (and MS DOS and PC DOS and their ilk) are splendid for dedicated computerists, but abyssally bad for those who simply want to use a computer. In contrast, the editor has, in a week, trained secretaries to usefully use Commodore machines. The difference lies in Commodore's invisible, unitary, ROM-based operating system, to which you speak English. In CP/M, you must learn Assyrian cuneiform, and switch (in those hieroglyphics) between programs to write, run, and edit, and to handle your disks.

With that original CP/M machine, the word-processing program was WordStar. When we tried to train a good word-processing operator (she'd used dedicated machies) on WordStar, she quit. We don't blame her; the command structure forced on WordStar by CP/M is so complex the screen must be globbed with menus (you can hardly see the text). Turn off the menus and you're lost. And you cannot change disks whilst editing; you get automatic backups (we don't want 'em), and automatic overwrites of old files (God, no!). And--you would not believe ED, the CP/M editor. If you've never tried it, do. Then go somewhere and weep. The editor in MS DOS is not much better.

Filenames are limited to eight characters. Can you imagine trying to find the file you want among 6000 different files, all indexed in eight characters? The eight-character limit still applies to CP/M variants and to MS DOS and PC DOS. Want a system with that sorry limitation in the age of megabyte memories?

We switched to Commodore. But--in a few years, Commodore machines will be Model T's. As with people who switched from horses to Henry's horseless carriage, we are happy to have them--they're simple and they work. But remember what happened to Henry. In 1920, he dominated the automobile market. By 1930, he was almost out of business. People traded up. Commodore may think it "makes computers for the masses, not the classes," but the masses acquire class tastes, as Henry found out. If Commodore expects to stay in computers, it must build something better than Model T's. We hope it does, having notions of trading up ourselves.

ARE YOU REDMARKED?

Look at the mailing label on this issue. If, underlined in red, you find a note that your membership has expired, this is your last issue. Please renew now if you're going to renew at all. Procrastination holds no pain for you, but ye ed just had the pleasure of re-entering by hand some 200 names and addresses of members whose renewals were late. Our mailing list program had deleted them all when dues were three months overdue.

Since we're not paid to edit or to redo mailing lists, our labor is clearly involuntary servitude and contrary to three Constitutional amendments we can think of and a whole bunch of statutes. If this keeps up, we'll sick the Feds on you.

ONCE OVER LIGHTLY
Miscellany

If you want to print APL on DIABLO 630 (Commodore 8300P), don't buy XEROX's metal wheel (9R 21135); the set of characters used is, here and there, wrong. The DIABLO plastic wheel (38150 01, APL 10) matches and prints the screen character set. We learned of the difference only after this issue was typed (and had to print it again).

While on wheels: a lot of DIABLO wheels substitute the degree mark for the apostrophe on the keyboard (SHIFT 7), including PRESTIGE ELITE, and are useless. Two wheels match the keyboard nicely: DIABLO's COURIER 72 (38107-01), 10-pitch; and ELITE 12 (38103-02) in 12-pitch. The latter prints the Gazette. You can interchange these two wheels and print the same characters (except for pitch). Both handle all printable program characters in any language but APL. For text work, using WordPro or 'CLIP, the DIABLO wheel COURIER LEGAL 10A provides a versatile character set, including the degree-, paragraph-, and section- signs. It's one of the few which lets you print almost anything printable if you master the key-substitution method in 'CLIP and WordPro. We tried over 20 wheels before we settled on the ones above.

If you're thinking about Commodore's new lettery-quality printer, suggest you wait a while. A couple of dealers report a lot of returns and a lot of problems. We've used QUME, SPINWRITER, and DIABLO. DIABLO is built like a Mack truck and has run for 2.5 years without any problem. Can't say as much for the others.

SHAME ON MICRO! MICRO magazine in issue 68 joined in the ongoing massacre of the English language. In an otherwise fine article on Networks (read it), the author talks of "routines that handle data compaction, encrytionation...." Ye Gods! Poor Will S. must now roll over in his grave again. Rotationation? Such polysyllabic crap is designed to impress others, but draws only laughter from anyone literate. In plain English, the polymush above translates to: "routines that compact and encrypt data..." We note the same article "collects together" every page or so. Ever hear of "collecting untogether?"

SASE PLEASE! Last month, we got over 300 letters needing reply, and two return envelopes. We enjoy your letters, but we don't enjoy addressing the replies or licking the stamps. Please be thoughtful; send a self-addressed postpaid envelope. (Associate editors, please ignore.) And do send envelopes big enough for a reply, not those tiny things sold to midgets for mailing recipes. Canadians: slip in a couple of those spendable dimes. (The Feds will probably get us...)

PET-COM Ph.D. Associates, Inc. of Toronto has forwarded a manual and disk so we can test PET-COM, a telecom package recommended by our members in TPUG. We'll try to have a software review next issue.

OP SYSTEM DISASSEMBLY
and mED REASSEMBLY

Some time back, we got a pretty well commented disassembly of the SuperPET operating system from John Toebes of Raleigh, and made copies for those we knew would be interested. The disassembly is over 200 pages long, and useful only for those who understand assembly language well. Anyone who wants a copy can get it on TPUG disk ST8, or from the editor at PO Box 411, Hatteras, N.C. 27943, for $6.00 in 8050, or $12.00 in 4040 (takes three disks in 4040). John also disassembled the microEDITOR and reassembled it (except for a few glitches). We sent a copy to TPUG, and Bill Dutfield called to say he'd removed the bugs and the package now assembles properly. His disk is here. Those who want to add some features to the mED (how about word-wrap and text move?) now have a clean base from which to start. If you want a copy, send $6.00 for 4040 or 8050. Postpaid. State your format. No SASE required. Yes, the reassembled mED works.

TOO SOON OLD, TOO LATE OBSERVANT For two years we've printed selected lines from the mED by entering the line numbers, as in 1200,1208 p ieee4. We finally got through our thick head the notion that there must be an easier way, and then tried: .,+8 p ieee4--which, of course, puts all text from the current line (.) through 8 following lines to our printer. (The current line is the one the screen cursor's on.) Ah, well. Care to guess what happens with: .,-8 p ieee4?

DOSbug IN MICROPOLIS 8050s TOO: Martin Goebel of St. John's, Newfoundland, writes that the DOSbug which sometimes keeps Tandon 8050 drives from reading or writing at power-up also afflicts his Micropolis 8050. Martin cures the problem in BASIC 4.0. with the: HEADER "x",Dd command. To reduce the bangs and crunches of no disk at all, he puts a blank disk in the drive called, though it is not necessary--the purpose being to move the R/W head, not to HEADER a disk.

LOAD LANGUAGES FROM ANY DRIVE Gee, we thought you knew this: you can load anything from a language disk from drive 0, or even from a 2031, if you simply preface the load command, at menu, like this: disk.e (which loads the mED). You can even load from a different device number, say device 9, with: disk9/0.e. It works for all languages and all machine-language programs loaded from menu. We mention it because a couple of old hands wrote that, drive 1 being out of commission, they were out of business in 6809. No more excuses. Back to work.

STOP DOESN'T ANY MORE In Version 1.0 microBASIC (haven't checked the other languages for this), ASCII 3 (Stop) on a disk file indeed STOPs a disk read dead in its tracks. To our surprise, we found out that V1.1 does not have the problem. If you're converting ASCII 3 to another CONTROL before filing to disk, and then re-translating after a disk read, you needn't bother in V1.1.

NO, YOU DON'T HAVE TO HAVE TWO SWITCHES! If you have an early 3-board SPET, and can't get the two retrofit switches to control UD11 and UD12 ROM sockets, you can work around the problem. You can use UD11 for WordPro, PAPERCLIP, or any other ROM needing the $A000-AFFF address with no sweat. UD12 ($9000-$9FFF) is the problem, for a ROM there blocks off the upper 64 in 6809--if you didn't read the January '83 Gazette, p. 1-2. If you did, you'll know you can unload any ROM in that socket to disk, and thereafter load and use the program from disk as easily as you'd load any other program in 6502. Once you have unloaded the ROM, take it out of UD12 to free up the $9000+ address range. So stop writing letters complaining you can't get switches. Who needs 'em?

WHY SO MANY PAGES?? A year ago, we promised you 20 pages per issue every two months, but we've done better than that. Last issue we published 27. How come?? What we earn on disk sales subsidizes printing and postage for the Gazette and pays for our endeavors to find the 9 out of 10 missing SuperPET owners. Thought you'd like to know. (Check the number of pages this issue...)

SMALL BUG IN G-IEEE8-15 There's an occasional bug in 'g ieee8-15' when it is used to enter DOS commands from mED. The bug is harmless, but if you don't know about it, you can think you've crashed. Once in a while, after you give a DOS command, the red 'drive in operation' light doesn't go off. This happens most often with INITIALIZE or VALIDATE commands. Should it happen to you, hit the STOP key. The red light will go out; you will see a lot of '00, OK,00,00' lines on the screen. Delete them; neither your disk nor the screen file in mED will be harmed. And the DOS command will have been executed.

Which reminds us the STOP key also stops any load from disk in mED. If you want to identify a file quickly, issue the command to 'get' it, then hit STOP. Only the first few lines of the file will load or show on the screen.

◇◇◇

ANOTHER WAY TO READ FILES PROPERLY IN MICROFORTRAN

by Stanley Brockman
11715 West 33rd Place
Wheat Ridge, CO 80033

[Ed. In issue 8, P.J. Rovero noted a bug in reading mFORTRAN data files and suggested a few ways around it. This article shows another way.]

A third way around the bug in reading disk files, as noted on page 93 of the Gazette, is to use list-directed reads [e.g., 'read *, a,b,c' or 'read(35,*) a,b,c)']. List-directed (LD) reads require that the data items in a record (either from disk or screen) be delimited by commas and the end of the line. The comma requirement could be considered by some (like me) to be a bug in its own right, but commas make it possible to read all of the data on a line from 'text' files (the default Waterloo type). An LD read will read whatever is between delimiters and assign the values to the corresponding variables in the input list, converting the values to the types (real, integer, or character) of their respective variables, taking any decimal point into account, as necessary.

LD writes also exist. As stated on page 146, 'List-Directed Output' in the mFortran manual, Chapter 'F', numerical data is written into pre-defined zones, with a blank (carriage control) character transmitted in the first record position. The first position of each numeric zone is reserved for the sign of the data value and is either a blank or a minus. There is also an extra blank padded between each of the numeric zones, although the manual does not state this. No extra blanks are padded before or after character data prior to being written to the file.

```
 program list_directed
 open(30,file='test.file')
 do i = 1,6
   a = 10 * rnd(.0)
   b = 10 * rnd(a)
   c = 10 * rnd(b)
   if(i .le. 3) then
     write(30,35) a,',',b,',',c
     write(6,35) a,',',b,',',c
35   format(1x,3(f11.7,a))
   else
     write(30,*) a,',',b,',',c
     write(6,*) a,',',b,',',c
   endif
 enddo
 rewind(30)
 print*
 do i = 1,6
   read(30,*) a,b,c
   print*,a,b,c
 enddo
 close(30)
```

The program at left demonstrates both LD and formatted writes and LD (unformatted) reads. We write the random numbers both to the 'test file' and to the screen (file 6). The data we write to the screen loses its first character since it's interpreted as a carriage-control character; otherwise, it duplicates what is written to the disk file; the program writes to disk and screen in the same manner. Note that the first 3 records are formatted writes while the second 3 are are LD writes. When we rewind the file [Ed. mFORTRAN's 'rewind' positions the file so we read/write again at the start of that file], we read the file in LD and print it (in LD) to screen as the next 6 records. Note that LD prints, like LD writes, automatically pad a blank to the beginning of each record; 'print' implies output to the screen; 'write' is file-oriented (the screen may be a file). I also pulled the disk file into the mED (loaded alone, not in mFORTRAN) and show it below, left.

```
 stop
 end
```

Below: Contents of Disk File:
(Asterisks added to show margin)
First 3 records formatted writes;
second 3 records, LD writes.

```
*** .6884766,  1.3150024,  2.5680542
   1.6650391,  2.2915649,  3.5446167
   9.8342896,  3.1134033,  3.7399292
   .65612793,  1.2826538,  1.9091797
   1.4535522,  2.0800781,  2.7066040
   1.4779663,  2.1044922,  2.7310181
```

Below, a dump of the screen after the program has run. Asterisks added.

1) Program output written to screen:

```
** .6884766,  1.3150024,  2.5680542
  1.6650391,  2.2915649,  3.5446167
  9.8342896,  3.1134033,  3.7399292
  .65612793,  1.2826538,  1.9091797
  1.4535522,  2.0800781,  2.7066040
  1.4779663,  2.1044922,  2.7310181
```

2) Program output, LD read and LD printed to screen from disk.

```
**.68847660  1.3150024  2.5680542
  1.6650391  2.2915649  3.5446167
  9.8342896  3.1134033  3.7399292
  .65612793  1.2826538  1.9091797
  1.4535522  2.0800781  2.7066040
  1.4779663  2.1044922  2.7310181
```

Note that I 'chained' the arguments to the RND intrinsic function, which probably is not good practice, but for this example it did not hurt. I have noticed that successive calls to RND as it is in the subroutine aren't especially random; the values of each result may appear to be random but their differences from one value to the next are quite likely to be identical when RND is used with only 0.0 as an argument. [As an exercise, try a DO loop using RND(0.0) and take successive differences.] Not all the differences are identical, but even those departures are fairly regular. This suggests that RND(0.0) may use the internal clock as a seed and that the randomizing algorithm is not very sophisticated.

LD reads and writes are much faster than their formatted equivalents. Yet because you have no direct control of the format in which data are stored, numeric precision becomes less as the numbers grow smaller. Scientific notation (E-format) is used automatically when values become small enough, but precision may be unacceptable before that point. Example:

Try successively dividing a number such as 1.23456789 by 10, and use an LD print to view results. I prefer to use formatted writes when saving data in order to avoid data loss because of poor representation in the file. LD reads, on the other hand, recover data from the files properly, with no error (other than that possibly arising from system errors in representing numbers in internal floating point form).

◇◇

BUFFER PROBLEMS IN THE 2031 DISK DRIVE WITH FASTERM/NEWTERM

Very few people use the 2031 drive (really half of a 4040), but those who do are warned that the buffer-handling in that drive will cause characters sent to disk from either NEWTERM or FASTERM (from files downloaded) to be dropped--at about 255 byte intervals. The 2K buffer in the 2031 is handled, internally, in a different way than in the 4040, 8050, or the 8250. At the end of a 255-byte segment of received characters, the 2031 fails to send an ACKNOWLEDGE signal (saying it has received a character) until it arranges to fill another 255-byte sector of drive memory. During this period, any characters which come in to the serial port cannot be sent to the drive by SuperPET (the last character hasn't been acknowledged). Result: from one to three characters may be missing in the disk file at this point. This problem does not exist with any other drives. Terminal programs which are interrupt-driven (NEWTERM and FASTERM are not) do not encounter this problem. If anyone in ISPUG who registered for membership with a 2031 drive only wants to return the ISPUG master telecom disk for this reason, we'll refund his money upon return of the disk to the Editor, at PO Box 411, Hatteras, N.C. 27943. We stand behind the programs we sell, because we test 'em first. NEWTERM and FASTERM were tested thoroughly--but not with the 2031.

◇◇

GET # WHAT? IS THIS HYBRID REALLY TRUE? How'd you like to get input both as a string and as the ASCII ordinal? You can eliminate what you don't want as input with simple commands ('if a>47 and a<58' gets rid of all numerals); you can get an array without subscripts or the trouble of dimensioning one, in the exact order of entry. And you can 'get' it in a form in which it can be erased, deleted, or revised (yes, revise a 'get'!) and then finally confirmed with a RETURN (yes, use RETURN on a 'get'). We ran into it whilst working on something else; it became curioser and curioser. Turns out SPET stores 'a' (see example, left) as an array! It does not execute the loop (ary a bit) until a CR is entered, whereupon all values of 'a', in sequence, are printed in the loop. If you doubt that, note the print statement FOLLOWS the quit statement, but it still prints. Any 'a' not confirmed by a CR is ignored. We printed 79 characters on a line, hit RETURN, and printed all 79 ordinals (while the characters were on screen!)--characters and ordinals in one swoop, with no conversion back and forth! Whilst everything is accepted as input without nasty error signals, you can screen it and output only what you want. In short, maybe we have a way to get 'trash in, jewels out'. Take a look....

```
100 ! 'getwhat1'. Prints char to
105 ! screen and 'gets' ordinals.
110 open #5, 'terminal', inout
115 loop
120   get #5,a
125   if a=13 then quit
130   print a;' ';
135 endloop
140 reset : stop
```

◇◇

SYS CALLS FROM LANGUAGE We've had some inquiries on how to reset end of user memory, load machine language modules, and use them from the various languages, so we give a short example of how to do it, below. There are two basic approaches: 1) run a short program from menu, which sets end of user memory [and also loads the ML program], then load the language, or 2), Load the language, and, before doing anything else, drop into the monitor and load the ML module. In either case, it's easiest if the ML module is written to perform two functions: first, to set memend_; second, to load the actual module to be used from the language. And it is easy to do. The very short program below demonstrates how to do it from menu, before you load a language:

```
xref printf_, putnl_
memend_  equ $22
service_ equ $32
                ;Section 1 : Load Module, Return to Main Menu

main     equ *             ;We define the start of program as MAIN. The built-
         ldd #main         ;in counter in assembler/linker will assign the starting
         std memend_       ;address to *. In this case, it's $7f60, the 'origin'.
         clr service_      ;Set this to 0 for return to main menu.
         rts               ;And return there after module is loaded.

                ;Section 2 : The Language Module

         ldd #test         ;This part we call from language. It is loaded with
         jsr printf_       ;Section 1 from menu, but does not run because of the
```

```
        rts             ;RTS in section 1.

test    fcc "This is the test line called into language with a SYS.%n"
        fcb 0
        end             ;The 'test' string is printed only from language.
```

If the program above is assembled and linked (the .cmd file is at left), it may be printed to the screen in any SuperPET language with a SYS call to the address of the language module (Section 2, above). How do you get that address? Call the 'sys.lst' file, which is created by the Assembler, into the mED. We print part of that file below, to show what we mean, and have annotated the material to make it clear. The 'Memory Location' column shows locations relative to program origin. In this case, the program origin is $7f60, so the '0000' below, added to $7f60, shows the location of 'main.' The language module starts at 0008. Add that to $7f60--and you know your SYS should be made to $7f68. Simplicimus. [The .lst file is handy indeed.]

```
"sys"
org $7f60
include "disk/1.watlib.exp"
"sys.b09"
```

```
Line    Memory
No.:    Location:    Object Code:        Source Code:

 7      0000                             main    equ *
 8      0000         CC   00   00                ldd #main
 9      0003         DD   22                     std memend_
10      0005         0F   32                     clr service_
11      0007         39                          rts

12      0008         CC   00   0F                ldd #test     ;The language
13      000B         BD   00   00                jsr printf_   ;module.
14      000E         39                          rts
```

The method above works in all SuperPET languages. It does not, however, cope with ye ed's weak and forgetful mind. There we are, language loaded, and we forgot to load the module from menu.... So, here's a second way to load the module, either from the monitor at main menu, or from the monitor in any language which uses the mED (after the language is loaded). The 'xrefs' and the definition of string 'test' are left out of 'sysmon.asm', below, to save space.

```
main    equ *           ;We load and run this in the monitor with a:
        ldd #main       ;>l sysmon.mod
        std memend_     ;>g 7f60
        swi             ;We SWI (software interrupt) in the monitor.

        ldd #test       ;Again, the sys call for the language.
        jsr printf_
        rts             ;Return to language requires an RTS.
```

Again, if you look at the .lst file, you'll find the language portion starts at $7f66, so that is the address for your SYS call. While the program above is very simple, it shows how long and complex assembly-language routines can be loaded and called from the languages. Both programs above will run in any SuperPET language if the right SYS call format is employed [In mPASCAL, sysproc(32614); in mFORTRAN, i=sys(cnvh2i('7f66'); APL, ⎕SYS(32614); mBASIC, sys hex('7f66')--for the monitor version directly above. Be sure to not use CAPITALS in any textual material to come back into APL.

Warning: if you load the ML module in the monitor, after your language is loaded, be utterly sure you load the module from the monitor first thing. Do not load a program, define strings, or do any other work before you load the module. Example of troubles: strings are stored near top of user memory. If you should define one before memend_ is reset, thou wilt crash. Memend_ will remain at the set value of $7f60 until you leave 6809, or until you reset it. It's best to CLEAR memory (reset all pointers), go into the monitor, reset memend_ ($32) to $7fff, and then CLEAR again, to again reset pointers. Perhaps we're super careful, but we haven't crashed when we followed this procedure.

◇◇

STARTER-PAK DISK AND MANUAL AVAILABLE

Ever since the price of SuperPET dropped to $995 by mail order, letters flood in from schools and owners asking how SuperPET works and complaining the manuals don't tell you much. We can't cope with that amount of mail, so we wrote a manual which distills into 28 pages the essential things you must know when first you open the box and plug SuperPET in--handling the DOS 6809-side, talking to disks and printer, what those external switches do, how to load languages from one drive, printing directories, handling files, the ASCII codes that control SuperPET's operations, and such--in short, the basic things we've learned from two years of using SuperPET. We kept it informal and full of examples. The companion disk illustrates the manual and gets into more details in all the languages but COBOL (don't speak Swahili, either).

Included are eight Reference Sheets which summarize everything from use of ASCII codes in SuperPET, through 6809 DOS Commands, to search/replace in the mED. We asked Steve Zeller for an APL WS to handle input/output to and from disks, from screen to printers, and from disks to screen and printer; Steve came up with a simple jewel for the disk. Jim Swift's alphabetizing LOADER for APL is included, along with Reg Beck's DOS.SUPPORT (APL DOS work from a menu), as well as two ready-to-use versions of UDUMP (no assembly, no linking) which'll load either from menu or in the monitor--plus an alphanumeric directory sort which puts two columns to screen and/or disk and printer. We added a program from P.J. Rovero which gives a two-column disk directory from main menu. If you can't use SPET after going through this stuff, best swap for an abacus. Schools may copy both manual and disk as often as they wish.

Programs and tutorial on disk fill a 4040. The disk is available in 4040 or 8050 format, with the manual, for $15 U.S. If you want it, send a check made out to ISPUG to the Editor at PO Box 411, Hatteras, N.C., 27943. State format. We're happy to say every program is commented, and that there's a six-page index to programs which explains the purpose of each. Every program is filenamed to show the language or facility where it runs or can be read. Documented, by gum!

◇◇

ANATOMY I : EXPLORATIONS IN SUPERPET

A few months ago, we received a copy of a note from Dr. H.O. Pritchard of York University about editing data files which contained CONTROL codes (ASCII 0 through ASCII 31), and, later, that inveterate explorer, Gary Ratliff, submitted his findings on text compression in the microEDITOR. Ah, serendipity! The two problems, apparently unrelated, finally fitted together as do pieces of a jigsaw puzzle. Then, CompuServe's 140-character strings got into the act, and we finally solved the mystery of why 80-character lines from a disk file print to screen double-spaced in the languages, and yet are single-spaced in the mED.

We start with the fact that the CONTROLS from 1 through 13 control SuperPET—and if read directly from a disk file, perform their function when sent to the screen. If ASCII 12 is in a file, for example, it clears screen and homes the cursor. Waterloo therefore strips the CONTROL codes from 1 through 31 from any file loaded into the mED, and substitutes for it a space--ASCII 32.

Second, we must carefully distinguish between NAUGHT (or '') and NUL (ASCII 0); they are not equivalent. A careful test in any language, or in the mED, demonstrates this quickly, as we'll show. NUL (ASCII 0) is a special case. Waterloo apparently employs it as a token for end-of-line or Carriage Return (ASCII 13), and you can plainly see it if you enter the monitor from the mED, and dump the text buffer at $400-$450. Note that all lines end with 00, not with $0d, the carriage return. In addition, in any language, the blank spaces following the last printable character on a line are '', and neither NUL nor ASCII 32.

Third, the languages and the microEDITOR do not print long strings in the same manner. You can demonstate this easily if you create a string of over 80 characters in language and then put it to disk. If retrieved from disk in language and printed to screen, the 81st and subsequent characters print on the next screen line. In the microEDITOR, that same disk file will print only the first 80 characters. If you delete those first 80 characters with a search/replace command, the remainder of the string will then appear. (The delete key will not perform this trick.) Why the difference? Let's sort this out, fact by fact, and later weave it into a useful pattern.

We demonstrate the effect of NUL (ASCII 0) by creating a disk file of a$, as is shown at left, with NUL in the middle. We open and print the file to screen in language, and see the second line at left; NUL prints as a small square. Then leave language, and pull the file into the mED, loaded alone, and surprise, suprise: you'll see the third line at left. Neither ASCII 0 nor the last half of the string will print. The reason: ASCII 0 (NUL) demarcates end-of-line, as we said. Repeat the exeriment with NAUGHT ('') substituted for NUL, and the whole string prints in both language and in mED. Note that you can search the strings above for a terminal NUL (or for ASCII 13) and never find one. It seems a NUL not only deletes what follows but also deletes itself. And the same thing is true in Assembly language; 00 marks end-string.

```
a$="Start"+chr$(0)+"end"

Start□end   [in language]

Start       [in mED]
```

What of disk files? We block-read the files above (SEQ), and found end-line and EOF marked by carriage returns, not by NULs. SuperPET obviously converts those NULs we see in the text buffer to CRs for disk files, and then reconverts them when it recovers the files from disk. Most curiously, the mED does not store any NULs in the files it creates for its own use, as Gary Ratliff shows below.

At this point, we can reach two conclusions: 1) Never use NUL in a disk file to be read in the mED. It can destroy part or all of a line, and 2) we still don't know fully what is going on. So, let's turn to Gary Ratliff for part of the answer. We'll weave the threads together at the end.

* * *

ANATOMY 2 : The Structure and Method of Text Compression in the MicroEDITOR

by Gary L. Ratliff, Sr.

I'm one of the vanishing breed of hackers who has a computer primarily to explore how it functions. In this article, we'll see how text files are stored in the microEDITOR. You'll have to use the V1.1 microEDITOR, which supports a MONITOR call from the mED (V1.0 does not).

Load the mED alone, from menu, and immediately enter the monitor; at the prompt enter: >d 0a00.20 to dump the first 32 bytes of user memory. You should see the line at left. We may safely conclude that 02 01 marks the start and end of the file. Quit the monitor and enter a single 'a' at left margin, between <beginning of file> and <end of file>; then return to the monitor, and dump again. You should see the line below. We gain a further insight into the 02 01 pairs; obviously the $61 is the ASCII code for 'a'. But what does the '03' in the third byte mean? And why the '02' after $61 (our 'a')? For the next experiment, we'll use a sequence of a's, as shown below. I've commented the code to the right of the a's to show what happens:

```
0a00  02 01 02 01 aa aa aa aa
```

```
0a00  02 01 03 61 02 02 01
```

	Bytes in line:	Characters:		Back Pointer:	
<beginning of file>	02			01	Start file.
a	03	61		02	
aa	04	61	61	03	
aaa	04	03	61's	03	
aaaa	04	04	61's	03	
<end of file>	02			01	End file.

```
>d 0a00.18
;0a00 02 01 03 61 02 04 61 61 *...a..aa
;0a08 03 04 03 61 03 04 04 61 *...a...a
;0a10 03 02 01 aa aa aa aa aa *........
```

To the left is the monitor dump from which the table above was constructed.

It seems to be clear. The beginning 02 01 and terminal 02 01 mark the start and the end of the file. The 02 defines the number of bytes in the starting line, and the 01 points back to the 02 as the start of the line. Nothing being between them, the line is blank. We notice also that as soon as there are more than two "a's", the number of repeated characters is shown by the preceding CONTROL code! But are we sure of our conclusions? Let's try another experiment, with a blank line between text lines, and full 80-character lines, as shown below:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
>d 0a00.18
;0a00 02 01 08 1f 78 1f 78 12 *....x.x.
;0a08 78 07 02 01 08 1f 78 1f *x.....x.
;0a10 78 12 78 07 02 01 aa aa *x.x.....
```

This is the monitor dump for the three lines shown immediately above. Data in hex. ASCII $78='x'

<u>Comments on the code:</u>

```
Start of file:     02      01

Line of 80 x's:    08      1f       78       1f       78      12       78       07
 (in hex)          |_____________back pointer is 7 bytes____________|
Comment:                   31       x's  +   31       x's  +  18       x's = 80 x's
 (in decimal)       8 bytes in the line.

Blank line:        02      01        Since nothing appears between the pointers,
Comment:         Bytes in  Back      the line is blank.
                 line.     pointer.
```

Second line: Same as first. End of File shown by terminal 02 01.

Why is the limit on repetitions of a character 31 (as it is above)? Why not use the exact number, which is 80 ($50)? Modify the code and try it. If you get a 'P', don't be surprised. The ASCII code for 'P' is $50. Waterloo very obviously uses the CONTROL codes from 1-31 in the microEDITOR for counting since they are easy to recognize and cannot be printed.

With the coding scheme shown above, finding lines in the mED is simple. The 1st byte at start-of-file (02) points to the start of the line of 80 x's, and that line starts with a pointer of 08 bytes, which points to the next line, and so on down the file. It's a simple chain, either additive or subtractive, and shows why the mED finds lines so easily. If the search/replace method in mED were to ignore lines shorter than the search string, or the remainder of a string when the remainder was shorter than the search string, searches for long strings just might be faster than those for short ones--a nice project for somebody.

You may ask, "What is practical about all this?" First, because you understand how text is compressed, you'll realize that structured programming and proper indentation do not waste precious memory (some word-processing programs indicate a blank line with 80 spaces). Tabs and other identations use memory sparingly; as an example, the line at left, indented four spaces, uses only five bytes: 05 04 20 61 04, and of those five, only two are used to show the indentation (04 20--$20 being ASCII 32, or a space). One programmer I know took out most of the indentations in his program to save memory. If he indented only 1 space, he saved 1 byte per line. Otherwise, he saved nothing. That knowledge is practical indeed. Second, we now can interpret text files in the monitor with some understanding of what we see. Third, we better understand the search/replace process and why it might be possible to speed it up. Fourth, we now know why it was so simple for Waterloo to give us the ability to insert blank lines in the mED, and to delete entire lines. Fifth, we've defined the framework within which a programmer must work to modify mED for word-wrap or for text-move. Since we now have all the code for the mED, I hope someone does it.

```
    a
```

Last, if comfort is practical, we are comforted by having explored and, I hope you agree, conquered, another aspect of SuperPET.

*   *   *

USE OF THE LESSONS   We're now able to apply the information above to several SPET problems. While writing this, we got a note from Don Momberg, of Green Brook, N.J., saying that COMPUSERVE had a nasty habit of sending him strings up to 140 characters long; his terminal program printed them to screen okay, but the file saved to disk and recalled into mED showed only the first 80 characters of such lines. He asked why. Isn't it now obvious? The mED does not print a new line to screen until it reaches the end-of-line defined by that first pointer of bytes-per-line. The solution is simple: amend the file by program: stuff in a CR at position 81 on long lines, and make the excess characters (those beyond 80) a following line. We wrote and tested such a program in ten minutes. Key lines are at the left. Anyone having trouble with long lines received in disk files from CompuServe or anyone else now has a solution.

```
...  ! line$ is original file line
150  transfer$=line$ : long=len(line$)
160  if long > 80
170    transfer$=line$(1:80)
180    excess$=line$(81:long)
190  endif
200  print #12, transfer$
210  if excess$ > '' then print #12, excess$ : excess$=''
```

Second, there's a problem betwixt the languages and mED on 80-character lines. If a disk file holds 80-character lines, those lines are double-spaced on output to screen from any printfile program in language, and from the 'type' command in microBASIC. Yet the same lines, if put to printer with a printfile program, or read on-screen in the mED, are single-spaced. On screen, in language, you never can tell a double-spaced line from an 80-character artifact. The cause: all the languages automatically CR and linefeed at screen column 80, and then encounter the terminating CR (or NUL), which sends another linefeed and double-spaces the lines to screen. Since the mED does not linefeed until it gets to the end of a line (however long), mED single-spaces those same lines.

Solve the problem in languages (screen output) by sensing an 80-character line; then print to screen an ASCII 11 without a CR before you print the next line. A disk file so handled will give consistent and true output to a printer, to screen from a printfile program, and in mED. The two lines at left are in every printfile program we use to solve the puzzle of "is that text really double-spaced, or is it an 80-character artifact?"

```
...
110 print line$   ! ASCII 11 is up-cursor.
120 if len(line$) => 80 then print chr$(11);
... ! Semicolon on line 120 essential.
```

So, a little work on anatomy indeed has practical results. And more next issue.

◇◇

BROTHER JOHN, BROTHER JOHN Marvin Cox, of 4900 W. 96th St., Oak Lawn, IL 60453, stimulated by Gary Ratliff's notes on music in SuperPET, sent in the following microPASCAL program, which repeats the nursery rhyme, "Brother John", in right smart fashion. If somebody knows how to turn up the volume, please write. We had to borrow a stethescope from a doctor (who thinks we're bonkers) to hear it. Note how easily Marv passes parms to procedure u, and those clear, readable but long variable names. Next issue, we'll hear from Gary Ratliff on why they run as fast as short variable names in SuperPET. Waterloo found a clever way to do it.

```
program music_pd(output);
var
  n:integer;
procedure u(pitch,duration:integer);
  var
    lengthofnote:integer;
  begin
  for lengthofnote :=0 to duration do
    poke(59464,pitch)
  end;
begin
  poke(59467,16);                {to activate, poke 16 into hex e84b}
  n:=15;
  repeat
    poke(59466,n);  {poke in any integer from 1 to 254 to change tone.}
    u(90,10);u(81,10);u(72,10);u(90,10);
    u(180,10);u(162,10);u(144,10);u(180,10);
    u(72,10);u(68,10);u(60,20);
    u(144,10);u(135,10);u(121,20);
    u(60,04);u(55,04);u(60,04);u(68,04);u(72,12);u(90,12);
    u(121,04);u(110,04);u(121,04);u(135,04);u(144,12);u(180,12);
    u(90,10);u(120,10);u(90,20);
    u(180,10);u(239,10);u(180,20);
    n:=n+35;
  until n>=254;
  poke(59467,1)           {to inactivate, poke in any integer except 16}
end..
```

◇◇

EXTENDED MONITORS and OTHER GOODIES Gary Ratliff's EXMON finally arrived; we have it on disk with Terry Peterson's beautiful SPMON, an extended monitor which lets you slowstep through a program, disassemble it, skip or run through subroutines, handle the DOS, save memory modules, and much more. By April 15, we'll have a bunch of assembled and linked assembly-language utility routines, fully commented, on the same disk. A few examples: pdir, which gives you a two-column directory from main menu, and optionally sends two-column directories to printer; four screen dumps, ready to go, which handle ANY printer and load either from menu or in the monitor (pick the one for your printer), and also optionally dump to disk; one that sends any SEQ file to printer from menu, an alphabetical sort, a machine-language program to change disk addresses from menu...and a bunch more. After April 15, you can get the disk in either 4040 or 8050 format (it fills a 4040) for $10 U.S. Write the Editor, and <u>state format</u>.

◇◇

IF EQ, IF NE, IF GT WHAT?? A Definition : CC Register Ever moan 'Strike 3!' when reading assembly language source code, when you see: 'if eq, if gt, or if lt,' etc., and find no <u>previous</u> comparison to tell you <u>what</u> is equal to, greater than, or less than <u>what?</u> Does the example at the left leave you wondering what is eq to what? Peace. During 'lda ,y+' the 6809 checks dutifully to see if the result in the A register is a zero. In the loop at left, we might be loading some values ending in a null byte (00). When that null loads, the Zero Flag of the Condition Code (CC) register is set to 1, which satisfies the program condition 'until eq.' The Waterloo Assembler manual defines all the Flags of the CC register clearly on page 93. It is in the CC register that you'll find a number of <u>specific conditions</u> reported. But--this leaves us with two questions: 1) How do you <u>sense or use</u> the conditions reported by the CC register, and, 2) How do you read the blamed CC register, anyway?

```
loop
  lda ,y+
until eq
```

On question 1): You sense the CC conditions using the excellent listing of condition codes on page 150 of the Waterloo Assembler manual. Then read Gary Ratliff's column, this issue, on when and how to use those codes. Next, you'll find that each operation of the 6809, as defined from page 102 of the manual forward, clearly tells you what happens down in the CC register if you use that instruction. Take our loop example, above. On p. 120 of the manual, you are told that a LD (load) for an 8-bit register will handle the CC flags as shown at left. Note we can expect the Zero flag (Z) to be set to 1 if all data bits are zeroes. So, when we said 'until eq' in the example at the top left of this page, we meant: 'until the zero flag is set.'

```
N - Set IFF bit 7 of data is set
Z - Set IFF all bits of data are clear
V - Cleared
```

With all this in hand, you know both what conditions you can set as tests, and what each 6809 operation will do to the CC register. But--now you face the second question: How the @*! do I easily read the CC register to find out what conditions are reported and what they mean? Peep into the monitor, and note that the CC register is reported in two hex digits, as in the example at the left. Use the short table below to convert $C9 to its binary equivalent. See how we distribute the binary values to define the status of each flag in the CC register in the line immediately below the table.

```
 CC
 c9
```

Handy-Dandy Conversion Table : Hex to Binary

Hex	Binary	Hex	Binary	Hex	Binary	Hex	Binary
0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

	$C				$9			
$C9 allocated:	1	1	0	0	1	0	0	1
Flag ID:	E	F	H	I	N	Z	V	C
Bit Number:	7	6	5	4	3	2	1	0
Full Flag name:	Entire State Flag	FIRQ mask bit	Half-carry Flag	IRQ mask bit	Sign Flag	Zero Flag	Over-flow Flag	Carry Flag
Logic operations affect?				No	Yes	Yes	Yes	No

Flags do not change until the processor executes an instruction which requires a change. Any bit can be changed by ORing or ANDing. OR CC with 1 will set flag if not set; AND CC with 0 will clear a flag if set.

BITS BYTES & BUGS : by Gary Ratliff, Sr.

215 Pemberton Drive, Pearl, Mississippi 39208

The major feature which distinquishes a computer from other equipment is its decision-making capability. In the 6809, you find enough types of decision-making branch instructions to confuse the unwary. Some are designed to test two's complement numbers and others test unsigned numbers. In this issue, we'll concentrate on how to select the branch instruction which will conduct the exact test you want.

First, exactly what is the difference between a two's complement number and an unsigned number? In an 8-bit computer we have that many bits to represent a number, identified from right to left as bits 0 through 7. Bit 0, on the right, is the least significant (LSB); bit 7 is the most significant (MSB). With this numbering scheme, the bit position represents the power of 2 to which a bit in that position is raised, as shown in the example below. This works well for unsigned numbers. But how do we discriminate between a positive and a negative value? The convention is that the MSB represents the sign of a number. If bit 7 is 0, the number is positive; if 1, the number is negative. When the MSB is a sign bit, numbers are in two's complement form. How are they expressed?

Value of Bit, if Set:	128	64	32	16	8	4	2	1
Bit (and power of 2):	7	6	5	4	3	2	1	0
Sample binary number:	0	1	0	1	0	1	0	1
		64	+	16	+	4	+	1=85

I convert 3 to -3, as two's complement, below. Step 1 changes all 0's to 1's and vice-versa. Then we add 1. The result is the two's complement, and the expression of -3 in binary. This is no fun and people detest having to do it. The instruction set of the 6809 will do it for you. To get the one's complement of the value in A register, use COMA. To

```
 3, unsigned:              00000011
 3, one's complement:      11111100
   add 1                          1
-3, two's complement:      11111101
```

get the two's complement, use NEGA. The 6502 doesn't have native instructions to do this; we get the one's complement with: EOR #$ff, and the two's complement by using CLC EOR #$ff ADC #1. Here, writers who came to the 6502 from the 8080, the Z80 or 6800 usually say 'unfortunately, the 6502 lacks the complement instruction.' They fail to mention that (fortunately, with pipelining), the 6502 often executes the instructions above faster than chips with a built-in 'complement', so don't be surprised if you find a Z80 running at 8 MHz so that it can keep up with a 6502 executing at 1 MHz. (I admit to a little bias....)

Now that you have the idea that numbers are represented in unsigned form if considered to be positive and in two's complement form when considered negative, let's explore how we can test numbers and make branch decisions in 6809.

I have created a table which expresses test conditions; it also shows which of the tests is best for our use. In all examples, I use > to mean greater than, => to mean equal to or greater than, =< to mean equal to or less than, < for less than, <> to mean not equal, and = to mean equal to. BR means 'branch.'

UNSIGNED TESTS:

Condition:	6809 Branch Mnemonic:	Structured Form:
Register = Memory	BEQ (BR if Equal)	if eq
Register <> Memory	BNE (BR if Not Equal)	if ne
R => M	BCC or BHS (BR if Carry Clear, if Higher or Same)	if cc or if hs
R > M	BHI (BR if Higher)	if hi
R =< M	BLS (BR if Lower or Same)	if ls
R < M	BCS or BLO (BR if Carry Set, if Lower)	if cs or if lo
M = R	BEQ (BR if Equal)	if eq
M <> R	BNE (BR if Not Equal)	if ne
M => R	BLS (BR if Lower or Same)	if ls
M > R	BCS or BLO (BR if Carry Set, if Lower)	if cs or if lo
M =< R	BCC or BHS (BR if Carry Clear, if Higher or Same)	if cc or if hs
M < R	BHI (BR if Higher)	if hi

SIGNED (TWO'S COMPLEMENT) TESTS

Condition:	6809 Branch Mnemonic:	Structured Form:
R = M	BEQ	if eq
R <> M	BNE	if ne
R => M	BGE	if ge
R > M	BGT	if gt
R =< M	BLE (BR if Less than or Equal to 0)	if le
R < M	BLT (BR if Less Than 0)	if lt
M = R	BEQ	if eq

M <> R	BNE	if ne
M => R	BLE	if le
M > R	BLT	if lt
M =< R	BGE	if ge
M < R	BGT	if gt

MISCELLANEOUS CONDITIONS

Condition:	6809 Branch Mnenomic:	Structured Form:
IF N BIT SET	BMI (BR on Minus)	if mi
IF N BIT CLEAR	BPL (BR on Plus)	if pl
IF V BIT SET	BVS (BR on Overflow Set)	if vs
IF V BIT CLEAR	BVC (BR on Overflow Clear)	if vc
ALWAYS BRANCH	BRA (BR Always)	n/a
NEVER BRANCH	BRN (BR Never)	n/a
SUBROUTINE	BSR (BR to Sub-Routine)	n/a

Structured programming statements and branching provide powerful means to make decisions. The tables above should let you perform the correct test for the values you compare, whether they be signed or unsigned, or when you are interested in a value at a specific memory location or in the value of a register.

◇◇◇

A WAY TO LOCATE RECORDS IN RELATIVE FILES BY DATE

Some of those who read my first article may have wondered where the "speed and ease" of using relative files comes in, since you have to know the record number to access the record. Here's a function which calculates record numbers for you, intended for files in which there's a record for each weekday. It can be modified to be usable if there is a record for every day. Give it a date; it gives you the corresponding record number.

```
800 def fn_reccalc (date_$, origin, wkday)     ! works for 20th century only
810 y_=1900+value(date_$(7:8)):m_=value(date_$(1:2)):d_=value(date_$(4:5))
820 wkday = 6 - wkday : weekend_ = 0
830 if int(m_)>2
840    m_=m_+1
850 else
860    y_=y_-1:m_=m_+13
870 endif
880 dy_=int(365.25*y_)+int(30.6*m_)+d_-694038  ! dy_=# days, 01/01/00-date_$
890 nday_ = dy_ - origin : if nday_<0 then print"Date before origin" : stop
900 if (nday_-wkday)/7 - int((nday_-wkday)/7) < 0.2 then weekend_ = 1
910 if nday_ < wkday       ! nday_ = # days from first date in file to date_$
920    fn_reccalc = nday_ + 1
930 else
940    fn_reccalc = nday_ - 2 * (1+int((nday_-wkday)/7)) + 1
950 endif
960 fnend
```

The heart of the function lies in lines 830-880, which calculate the number of days from Jan 1st, 1900 to the date, 'date_$', which you supply. The second parameter, 'origin', is the number of days from Jan 1, 1900 to the <u>first day</u> in the file. The third parameter, 'wkday', is the day of the week on which the first day in the file falls (1=Monday, 2=Tuesday, etc.).

When you set up a file, you must determine 'wkday' from a calendar and then calculate 'origin'. You can do the latter by using the function itself. Suppose the the first day in your file is 6/11/82. Type the line shown left, below, in immediate mode. Note the format of the date, which MUST be expressed as an 8-character string, in the form MM/DD/YY. (The other two parameters do

```
x=fn_reccalc("06/11/82",0,1)
```

ue of 'origin', so type 'print dy_' in immdiate mode. You'll get 30112 as the value of 'origin' (the number of days since Jan 1, 1900). And, since 6/11/82 was a Friday, wkday=5.

Now let's put 'origin', date, and 'wkday' to work. Suppose you need to determine the record in which to put data for 12/7/82. Enter the immediate mode line shown at left, and you will be told to put the data in record 128. Obviously, 'origin' and 'wkday' are constants for a particular file, so I usually store them in record #0, which is not used for data. Incidentally, if you accidentally specify a weekend date, the variable 'weekend_' is set to 1 (true).

```
print fn_reccalc("12/07/82",30112,5)
```

As written, the function works for this century only. If modified slightly, it should be usable until Feb. 28, 2100, though I have not checked it beyond 1999. To use the function for dates after 1999, you have to supply dates as a 10-character string ("MM/DD/YYYY") and alter line 810 to read:

```
   810 y_=value(date_$(7:10)):m_=value(date_$(1:2)):d_=value(date_$(4:5))
```

If you need to store data for every day of the week, not just weekdays, modify the program as shown below. Example: let the first day be 8/5/82. So type your call: x=fn_reccalc("08/05/82",0), and print 'dy_' for 'origin', which turns out to be 30167. Then type: 'print fn_reccalc("12/31/82",30167)' to learn that data for 12/31/82 should be placed in record #149.

```
Delete lines 900-950 and line 820
Modify line 800 to read '800 def fn_reccalc(date_$, origin)'
Add '900 fn_reccalc = nday_ + 1'
```

APL CHARACTER SET FOR EPSON PRINTERS

The three articles following show how to print the APL character set to Epson printers. They use two distinct methods to create the character at printer after the character set is defined. In the first, the dot-matrix pattern for the character is transmitted for each character printed. In the second, the character set is downloaded to printer RAM and the characters are called with their normal ASCII code. Terry Peterson uses method 1 on the MX-80, to which you cannot download a character set. Reginald Beck uses method 2 to download a set to the FX-80. Steve Zeller does it both ways. All have created character sets which <u>should</u> work, if properly employed, with any Epson printer. Those who own Commodore printers will find in all articles clues on how to approach the problem.

All programs and all character sets in the next three articles are on disk (including the character sets or their representations), so that you need not design your own unless you care to. See end of articles on how to get the disk.

All characters sent to Epson printers are defined as the position of dots in 11 vertical slices of an 8-by-11 matrix, as shown at the left, below. There, we de-

```
              Column              Value
Row     1 2 3 4 5 6 7 8 9 0 1    of Col
     1    *   *   *   *   *       128
     2    *       *       *        64
     3            *                32
     4            *                16
     5            *                 8
     6            *                 4
     7            *                 2
     8            *                 1
```

define the character 'T' as an example. The columns (vertical slices) are each defined by a power of 2, assigned to the rows of the column. Column 6, for example, is the sum of all the digits in the column 'Value.' Column 1 is 0; column 2 is 128+64=192; column 4 is 128. By using powers of 2, each sum sent is utterly unique, and can be parsed as one and only one column pattern. If you define your own set, this is how and where you start. Note: watch out for ⎕IO. In both Terry's and Steve's listings, ASCII 0 is defined as 1 in the character set, with ⎕IO set to 1. You must subtract 1 from the ASCII code and 1 from the column totals in the listings to convert to the matrix values above. Example: col. 2, above, will show a value of 193 in listings, when its base 0 value is 192.

Last, note that while dots may be adjacent in columns, some Epson manuals say to leave a space in rows, as we've done above. Both Terry and Steve intentionally ignored this, and report no problems. Suggest you check on your printer before you cram dots together in each column of one row. (Reg Beck left row space.)

◇◇

[Ed. Terry's work below was done in V1.0 of APL. If you want to use the method in V1.1, the programs must be revised. We publish the article because it shows one way to solve the problem of printing the APL character set. Basically, Terry uses the dot-matrices of the SuperPET character generator ROM. He unloads that character generator, and converts the matrices to generate the set at printer. The final character set is available on disk.]

LISTING MICROAPL TEXT USING DOT-MATRIX PRINTER "BIT-GRAPHICS"

by Terry Peterson, 8628 Edgehill Court, El Cerrito, CA 94530

Printing APL listings is a perennial problem. Every new APL user has to find a way to get hard copies of nonstandard-character APL text. Typically this problem has been solved by using a special, dedicated-to-APL printer or, with the advent of removable print elements, by fitting one's printer with a special APL printing element. The existence of dot-matrix printers with programmable characters or 'bit-graphics' makes possible a new solution to this problem for the APL programmer. This article describes the procedure I used to obtain a data representation of the SuperPET's microAPL character set suitable for use in printing APL listings on an Epson MX-80 with Graphtrax 80 or Graphtrax+. It also presents the resulting data set and the APL functions I wrote to produce APL listings. One should be able to use this procedure and these APL functions, with little modification, to produce APL hardcopy on any dot-matrix printer having dot-programmable printing capability.

The characters of any character set on a dot-matrix printer or CRT are made up of arrays of dots. In the case of the SuperPET each CRT character is composed of an 8-by-8 matrix of black or green dots. Each dot location is called a 'pixel.' To form a given character, a pattern made from the 64 pixels in that character location is projected on the screen under the direction of an integrated

Listing 1

```
[ 0]  CHAR_XFORM ;A;Z;⎕IO
[ 1] ⍝
[ 2] ⍝      THIS FUNCTION READS THE 'PROGRAM' FILE
[ 3] ⍝      NAMED 'S.P.CHROM' (IN ASCII UPPERCASE)
```

```
⍝      THAT WAS 'SAVED' FROM THE PET M.L. MONITOR
⍝      AND WHICH CONTAINS AN IMAGE OF THE SUPERPET
⍝      CHAR. GEN. ROM.  IT FIRST SKIPS OVER THE
⍝      TWO CBM AND THE ASCII CHARACTER SETS.
⍝      THEN IT READS THE FOURTH SET IN 8-BYTE (ONE
⍝      CHAR.) CHUNKS, TRANSPOSES ROWS AND COLUMNS,
⍝      AND WRITES THE TRANSPOSED BYTES TO THE FILE
⍝      'APLCHRS' FOR USE IN 'MX_BYTES'.
⍝
         ⎕IO←0
         'DISK/1.Γ.*.∩∆Po|,*P∇' ⎕TIE 99   ⍝OPEN '1:S.P.CHROM,PRG'
      →  (0≠⍴⎕STATUS 99)/0                ⍝QUIT IF NOT FOUND
         'APLCHRS' ⎕CREATE 98             ⍝FILE FOR TRANSFORMED BYTES
         Z ← ⎕GET 99,2+(3×1024)           ⍝SKIP LOAD ADDRS + 1ST 3 SETS
   LOOP: Z ← ⎕GET 99 8                    ⍝GET 8 BYTES = ONE CHAR
         ⎕ ← A ← ⍉(8⍴2)⊤⎕AV⍳Z             ⍝TRANSFORM + DISPLAY
         (⎕AV[2⊥A]) ⎕PUT 98               ⍝WRITE XFORMED BYTES
      →  (0=⍴⎕STATUS 99)/LOOP             ⍝LOOP UNTIL DONE
         ⎕UNTIE 99
         ⎕UNTIE 98
```

circuit called a CRT controller (CRTC). The characters that are to be displayed are in turn read by the CRTC from an area of memory called screen RAM. A complete description of arbitrary dot patterns in the 64 pixels of the 2000 character locations on the SuperPET screen would require at least 16,000 bytes of RAM—a rather huge amount of memory to dedicate to a display (but that's what the IBM PC and Victor computers, etc., do for graphics only). If, however, we restrict ourselves (as does the SuperPET) to a description of 2000 characters selected from a set of no more than 256 characters, the minimum size of the screen RAM may be reduced to equal the number of characters.

This economy is accomplished by having the CRTC 'look up' the actual 8-by-8 pixel pattern for each character in a 'table' of patterns contained in a 'character generator' ROM. The ROM can contain a complete description of a 256-character set in 2K bytes. An additional economy is possible if the second half of the set is made to be the negative or reverse-field image of the first. Then the CRTC can calculate the 2nd 128 patterns from the 1st 128. Using this scheme, the 4K character generator in SuperPET contains 4 character sets of 256 characters, each set composed of 1024 bytes of 8-bit code defining 128 8x8 matrices of characters. The first two are the usual CBM character sets, the third is an ASCII set used by all Waterloo languages except APL, and the last is the mAPL set.

Listing 2

```
  BYTE_GEN
     ⎕IO ← 1
⍝ THE FOLLOWING DEFINES A VECTOR 'B' TO CONTAIN THE MX80
⍝ EQUIVALENTS OF THE COLUMNS OF DOTS SHOWN ON THE SUPERPET
⍝ SCREEN BY THE CHARACTER GENERATOR ROM. 'B' IS THEN WRITTEN
⍝ TO THE DISK AS A 'BARE' SEQUENTIAL FILE FOR LATER USE BY
⍝ THE FUNCTION 'MX_CHARS'.
⍝
  B←    1 1 16 16 16 16 16 1 1 1 1 256 1 1 1 1
  B←B, 17 17 17 17 17 17 17 17 17 17 17 241 1 1 1 1
  B←B,  1 1 1 241 17 17 17 17 1 1 1 32 17 17 17 17
  B←B, 17 17 17 32 1 1 1 1 17 17 17 241 17 17 17 17
```

[Ed. We do not print all of this listing, since it is on disk and would be tedious to copy. The abbreviated listing is shown to help you follow the other programs.]

Knowing the dot-by-dot definitions of a character set lets us display its characters on printers like the Epson MX series, since such printers permit us to specify the print lines in terms of individual columns of dots instead of character by character. (Some printers allow you to specify the pixel pattern [Commodore] or character sets [Base 2]). This facility is equivalent in function to the bit-graphics approach and conceptually more straightforward to implement. The details of character formation in that case, however, are a bit messier, since the characters are not usually 8-by-8 arrays.

Obtaining APL hard copy on bit-graphics printers is merely a matter of drawing the lines of text using the dot-patterns contained in SuperPET's character-generator ROM. Having realized this, one is left with two (minor) impediments: (1) The character generator ROM cannot be directly accessed from either of the two microprocessors in SuperPET, and (2) the bytes in the character generator describe <u>rows</u> of pixels while the printer description requires <u>columns</u> of pixels.

One may solve the problem easily: remove the character generator ROM (c.g. ROM), plug it into the expansion socket of another SuperPET, and save an image of that socket's address range on tape or disk. If you don't have access to a second computer or are squeamish about messing with SuperPET's innards, skip to the description of listing 2 (BYTE_GEN) for an alternative way. Only part of listing 2 is shown, so you can get the idea. The full listing is on disk. [Or use the Zeller/Beck method.]

Whether you copy your own c.g. ROM or use the disk material, you now have a copy of the character generator ROM that the microprocessor can see. Next, we address the row-column problem by rearranging the bits of our ROM image so we have bytes representing vertical columns of pixels. We do this with the APL function CHAR_XFORM we show in listing 1; it reads the APL character section of the c.g. ROM image one character (8 bytes) at a time. Each character representation is converted into an 8-by-8 array, which we then transpose, swapping rows and columns. The array is then reconverted to 8 bytes and written to the file APLCHRS. As this process continues, the 8-by-8 matrix representation of each character is displayed on the CRT to allay fears that nothing is happening.

BYTE_GEN, listing 2, gives you another way to obtain a disk file representation of the columns of the APL character set. If you have BYTE_GEN on disk, it produces a disk file identical to that you'd get by copying the c.g. ROM and executing CHAR_XFORM. BYTE_GEN defines a vector 'B' to contain the numbers corresponding to the pixel columns of the microAPL character set, and then writes 'B' to disk as a sequence of bytes. Whichever of the two ways described is used to generate it, the resulting disk file may be used in APL functions such as MX_DRAW, shown in listing 3, to produce hard copy listings of APL text on a dot-matrix printer. Notice that only 128 characters are defined in APLCHRS. I ignored the reverse field characters since they never appears in listings. These could be added, of course, for use with a screen dump routine.

```
[ 0]  MX_DRAW FUNCTION ;XREP;SHAPE;CMDSEQ;⎕IO;IC;J
[ 1]       ⎕IO ← 0                              ⍝USE ZERO ORIGIN
[ 2]       MX_CHARS                             ⍝GET CHAR SET BYTES
[ 3]       IC ←  (14⍴⎕AV[0]),'⍱⍲⍒⍋⌽⍉⊖⍟⍕⌶⌶⍀⍫⍝⍞!⌹⌶'    ⍝DEFINE INTERNAL CHAR SET
[ 4]       IC←IC,' ¨)<≤=>]∨∧≠÷,+./0123456789([;×:\'
[ 5]       IC←IC,'¯⍺⍳∩⌊∊_∇∆⍳∘''⎕|⊤○*?⍴⌈~↓∪⍵⊃↑⊂←⊢→≥-'
[ 6]       IC←IC,'⋄ABCDEFGHIJKLMNOPQRSTUVWXYZ{⊣}$$'
[ 7]       XREP ← ⎕CR FUNCTION                  ⍝MAKE CHAR ARRAY
[ 8]       SHAPE ← 1 8 × ⍴ XREP                 ⍝CALC. SHAPE OF OUTPUT
[ 9]       CMDSEQ ← ⎕AV[27,76,(256|SHAPE[1]),⌊SHAPE[1]÷256] ⍝<ESC>,L,NBYTES
[10] ⍝
```

Listing

3

```
[11]       'IEEE4' ⎕CREATE 99                       ⍝OPEN PRINTER FILE
[12]       J ← 0                                    ⍝INITIALIZE COUNTER
[13]       ⎕AV[15] ⎕PUT 99                 ⍝<SI> FOR NARROW CHARS FOR NUMBERS
[14] LOOP: ⎕AV[13] ⎕PUT 99                          ⍝POSITION PRT HEAD AT LEFT
[15]       (⎕XR '   ←',(2 0⍕J),'→ ') ⎕PUT 99        ⍝OUTPUT LINE NUMBER
[16]       CMDSEQ ⎕PUT 99                           ⍝SET PRNTR TO GRAPHIC MODE
[17]       MX_BYTES[IC⍳XREP[J;];] ⎕PUT 99           ⍝OUTP GRAPHIC IMAGE OF LINE
[18]    → ((J ← J+1)<SHAPE[0])/LOOP                 ⍝LOOP UNTIL ALL LINES DONE
[19]       ⎕AV[13 10 13] ⎕PUT 99                    ⍝FLUSH BUFFER + CANC. <SI>
[20]       ⎕UNTIE 99                                ⍝CLOSE PRINTER FILE
```

```
[ 0]  MX_CHARS
[ 1] ⍝         GET CHAR-GEN DOTS FROM DISK FILE
[ 2] ⍝         UNLESS MX_BYTES ALREADY IN WS.
[ 3] ⍝
[ 4]    →  (0≠⎕NC'MX_BYTES')/0      ⍝IF HAVE BYTES, QUIT
[ 5] ⍝
[ 6]       'APLCHRS' ⎕TIE 99        ⍝OPEN FILE WITH CHAR. DOTS
[ 7]       MX_BYTES←⎕GET 99 1024    ⍝GET THE CHAR.-SET DOTS
[ 8]       ⎕UNTIE 99                ⍝CLOSE FILE
[ 9] ⍝
[10]       MX_BYTES ← 128 8 ρ MX_BYTES  ⍝RESHAPE INTO CHAR.-BYTE ARRAY
```

Now that we have the necessary data representation of the APL character set, all we need is a transcription routine to convert the internal representation of APL text to the appropriate sequence of bytes to 'draw' on a printer. The routine, MX_DRAW, is given in Listing 3, which also shows MX_CHARS, a routine to retrieve the byte data from the disk file. MX_CHARS reads the character set bytes from the disk file 'APLCHRS' into the global variable MX_BYTES. MX_DRAW first calls MX_CHARS to make sure MX_BYTES has been defined, then it defines a character vector 'IC' to contain the internal representation of the characters of the microAPL set in the sequence obtained by poking the integers zero to 127 to the screen. (Note this is not identical to the first 128 elements of the APL 'atomic vector.') Since the first 14 characters are graphics or control characters, they are effectively ignored by defining them to be 'nulls' in IC. In line [7] the function to be listed is transformed by the intrinsic function 'canonical representation' into a character array XREP having as many rows as there are lines in the function definition and as many columns as there are characters in the longest line of the function definition. In line [9] the byte sequence is calculated that signals to the MX-80 that a bit-graphics image is coming. Then a file to the printer is opened and each line of the function (that is, each row of XREP) is output to the printer, following its line number, as a sequence of bytes selected from the rows of MX_BYTES by matching the characters in XREP and IC. I apologize to the APL experts reading this, as I am sure a purist would not use a loop such as in lines [14-17]. In my defense I submit that I did write a 'loopless' version of MX_DRAW, but it couldn't list itself because of a 'WORKSPACE FULL' error.

◇◇

GENERATING THE APL CHARACTER SET : PRINTING BY BIT-MAPPED GRAPHICS AND BY DOWNLOADING.

By Steve Zeller, 6425 31st St. N.W., Washington, D.C. 20015

You have three options for generating the APL character set with the dot-matrix printers most of us use: (1) to insert a chip which supports APL, (2) to print using bit-mapped graphics, and, most recently, (3) to download an APL character set to the printer. Some chips do exist for Epson MX-80 and DEC LA-120 printers, but I know of none for the CBM 4022 and 8023 models. This article presents an APL character set for Epson dot matrix printers. For the MX series of printers, printing is done in bit-image graphics mode. This requires extra routines in the WS to print, however, and is slow. The newer FX series allows downloading of a character set to the printer: APL printing is then accomplished at 160 cps. All the tools needed to develop and print APL characters are presented here. It is hoped that these tools will provide CBM 4022/8023 owners with a solution to their problem as well. Note that other character sets, such as the PET ASCII-Graphics set, can be developed with this approach. This material was developed simultaneously by Reg Beck and myself, while Terry Peterson did his work over a year ago in V1.0 APL.

In bit-image mode, eight of the wires in the Epson print head can be controlled by a single byte from the micro. Characters downloaded to the FX printers can be at most 8X11; I chose an 8x10 matrix, with the 11th column left empty, and ignored other features, such as true descenders and proportional spacing.

Terry Peterson based his design on information stored in the SPET's character generator. For several reasons, I decided to design the APL characters myself. It turns out, however, that the design problem is a nice application for APL. The character is first represented by an 8 by 10 matrix, say, of "nulls" on the screen. Using the SPET's cursor controls, I build a character using "quad's" and convert it to a boolean data matrix of the same size. Finally, the APL operator, "decode", is used to map each column of the representation matrix into the relevant byte for the printer. I show the overstruck character "del stile"

at the left, and then generate it on the screen by the methods explained below. Details of the screen "get" function can be found in Vol. 1, No. 8 (p. 106) of the Gazette.

```
      ∇GET_CHR[⎕]∇
[  0]    BMAT ← GET_CHR ;ANS;MAT
[  1]   CLEAR
[  2]   ⎕←8 10ρ'~'
[  3]   ⎕TC[⎕IO+6],'NOW CURSOR AROUND MATRIX AND BUILD CHARACTER WITH ¨⎕¨.'
[  4]   'WHEN FINISHED, CURSOR DOWN TO PROMPT AND HIT <RETURN>'
[  5]   ⍞←REVERSE '>'
[  6]   ANS←⍞
[  7]   MAT←ΔGETSCR[1+ι8;ι10]
[  8]   BMAT←MAT≠'~'
                |-------------------------------------------------------------
~~~~⎕~~~~~      |  THE CHARACTER IS TO BE A MATRIX OF 8 BY 10 DOTS. THE ROUTINE
~~~~⎕~~~~~      |  ABOVE FIRST CLEARS THE SCREEN AND THEN PRESENTS AN ¨EMPTY¨
~~~~⎕~~~~~      |  CHARACTER CONSISTING OF <TILDE>. USEING THE CURSOR KEYS, A
⎕⎕⎕⎕⎕⎕⎕⎕⎕~      |  CHARACTER IS DESIGNED WITH <QUAD>. WHEN DONE, CURSOR DOWN
~⎕~~⎕~~⎕~~      |  TO THE PROMPT AND HIT <RETURN>. HE RELEVANT 8×10 PORTION
~~⎕~⎕~⎕~~~      |  OF THE SCREEN IS CONVERTED TO A BOOLEAN MATRIX AND RETURNED.
~~~⎕⎕⎕~~~~      |  FOR EXAMPLE, WITH: IC←GET_CHR, THE MATRIX IS SHOWN BELOW.
~~~~⎕~~~~~      |-------------------------------------------------------------

NOW CURSOR AROUND MATRIX AND BUILD CHARACTER WITH ¨⎕¨.
WHEN FINISHED, CURSOR DOWN TO PROMPT AND HIT <RETURN>
>

NOW CURSOR AROUND MATRIX AND BUILD CHARACTER WITH ¨⎕¨.
WHEN FINISHED, CURSOR DOWN TO PROMPT AND HIT <RETURN>
>
```

```
IC
0 0 0 0 1 0 0 0 0 0       |-----------------------------------------------
0 0 0 0 1 0 0 0 0 0       | EACH COLUMN REPRESENTS A "FIRING PATTERN" FOR THE
0 0 0 0 1 0 0 0 0 0       | PRINTER'S EIGHT PINS. THERE ARE 2 TO THE 8TH POWER
1 1 1 1 1 1 1 1 1 0       | UNIQUE COMBINATIONS, AND THE EPSON EXPECTS AN ASCII
0 1 0 0 1 0 0 1 0 0       | CHARACTER TO TELL IT WHICH ONE IT IS. WE NEED TO
0 0 1 0 1 0 1 0 0 0       | KNOW WHAT THE BASE 2 VALUE OF EACH BOOLEAN COLUMN
0 0 0 1 1 1 0 0 0 0       | IS AND WE CAN DO THAT WITH "DECODE", AS SHOWN BE-
0 0 0 0 1 0 0 0 0 0       | LOW. THE FIRST COLUMN MAPS INTO 16 AND HENCE WE
      2⊥IC                 -----| NEED TO SEND THE 16TH CHARACTER IN ⎕AV TO THE
16 24 20 18 255 18 20 24 16 0    --|PRINTER IN ORDER TO FIRE THE PINS CORRECTLY.
                                   |-----------------------------------------------
```

Since I need to construct a full character set, it's worthwhile developing some other tools as well. The functions below allow me specify a range of characters (All 128 at one sitting is too much!). Each character is displayed on screen and followed by a prompt for the name of the character, which is stored in APLNAMES. The design, using the material above, is stored in APLCHARS and then sent to the printer. Note that the most compact storage of the bit image information consists of the relevant bytes and not their boolean representation matrix. The routine BIT_Epson puts the printer into dual density bit image mode and determines the length of the bit image line being sent to the printer (always the same in this application).

```
      ⍴APLCHARS
128 10
      ⍴APLNAMES
128 15
      ∇BUILD_CHARS[⎕]∇
[  0]    BUILD_CHARS ;N;I;IC
[  1]   ⍝MASTER ROUTINE TO BUILD APL CHARACTER SET FOR EPSON PRINTER
[  2]   S1:'ENTER: START AND FINISH ≠''S, (1-128)'
[  3]     →(2≠⍴N←⎕)/S1
[  4]   I←N[1]-1
[  5]   S2:'APL CHARACTER: ',⎕AV[I←I+1]
[  6]     'ENTER: NAME OF CHARACTER (15 CHRS MAX)'
[  7]     APLNAMES[I;]←15↑⍞                              |-----------------
[  8]     APLCHARS[I;]←⎕AV[⎕IO+2⊥IC←GET_CHR]             |∇OPEN_PTR[⎕]∇
[  9]     PRINT APLNAMES[I;],' ',BIT_EPSON APLCHARS[I;]  |   OPEN_PTR
[ 10]     →(N[2]>I)/S2                                    |  'IEEE4' ⎕CREATE 4
[ 11]   'DONE'                                            |∇PRINT[⎕]∇
      ∇BIT_EPSON[⎕]∇                                      |   PRINT MSG
[  0]    R ← BIT_EPSON STUFF ;⎕IO;NCOL;PCODE              |  MSG ⎕PUT 4
[  1]   NCOL←(¯1↑⍴STUFF)+⎕IO←0                            |∇CLOSE_PTR[⎕]∇
[  2]   PCODE←⎕AV[27 76,(256|NCOL),⌊NCOL÷256]             |   CLOSE_PTR
[  3]   R←PCODE,STUFF                                     |  ⎕UNTIE 4
                                                          |------------------
```

It is sometimes hard to anticipate how the bit image will actually appear on paper, however, so an editing capability is available in EDIT_CHAR; characters can be printed on an ASCII printer with PRINT_CHAR. A partial APL character set is shown in Table 1, to give you an idea of what the set looks like, together with the ten-bit image bytes for each character (origin 1).

```
      ∇EDIT_CHAR[⎕]∇
[  0]    EDIT_CHAR N ;BMAT
[  1]   ⍝EDITS EXISTING CHARACTER
[  2]   →((1>N)∨(128<N))/0
[  3]   CLEAR
[  4]   (⍕N),'-',APLNAMES[N;],':',⎕AV[N]
[  5]   8 10⍴(,(8⍴2)⊤¯1+⎕AV⍳APLCHARS[N;])\'⎕'
[  6]   ⍞←REVERSE '>'
[  7]   ANS←⍞
[  8]   BMAT←(∆GETSCR[(2+⍳8);⍳10])='⎕'
[  9]   APLCHARS[N;]←⎕AV[⎕IO+2⊥BMAT]
      ∇PRINT_CHAR[⎕]∇
[  0]    PRINT_CHAR N
[  1]   ⍝PRINTS OUT ASCII REPRESENTION OF CHARACTER
[  2]   →((1>N)∨(128<N))/0
[  3]   PRINT (⍕N),' ',APLNAMES[N;]
[  4]   PRINT 8 10⍴(,(8⍴2)⊤¯1+⎕AV⍳APLCHARS[N;])\'⊃'
```

Now having a character set, we may print it either of two ways. The first method is bit-image printing; shown below is one way to do this with an MX printer. The function PRINTAPL will send APL characters to the printer in bit-image codes. Screen "get" routines are useful to capture data before printing, or CONVFN will produce a visual representation of a function as a character matrix. Thus, PRINTAPL CONVFN 'yourfn' will produce a function listing. If space is tight in the WS (it usually is), the best bet is to send the APL output to a file and to print it later. Note that there is no need to "explode" the APL output into its external representation: overstruck characters can be printed with bit image control "as is". I used this method for several years with an MX-100, and it should work as well on the MX-80.

```
      ∇PRINTAPL[⎕]∇
[  0]    PRINTAPL STUFF ;I;NROW
[  1]   →(1<⍴⍴STUFF)/MATRIX
[  2]   PRINT BIT_EPSON ,APLCHARS[⎕AV⍳STUFF;],⎕AV[⎕IO]
[  3]   →0
[  4]   MATRIX:NROW←(1↑⍴STUFF)+I←0
[  5]   S1:PRINT BIT_EPSON ,APLCHARS[⎕AV⍳STUFF[I←I+1;];],⎕AV[⎕IO]
[  6]     →(NROW>I)/S1
      ∇CONVFN[⎕]∇
[  0]    CFN ← CONVFN FN ;NROWS;NOS;⎕IO
[  1]   ⎕IO←1
[  2]   →(3=⎕NC FN)/OK
[  3]   'NO FUNCTION...',FN
[  4]   OK:CFN←⎕CR FN
[  5]   NROWS←(⍴CFN)[1]
[  6]   NOS←(⍉(1,NROWS)⍴'['),(⍕⍉(1,NROWS)⍴(⍳NROWS)-1),(⍉(1,NROWS)⍴']')
[  7]   CFN←NOS,CFN
[  8]   CFN←((1,(⍴CFN)[2])⍴(⍴CFN)[2]↑'    ∇',FN,'∇'),[1]CFN
```

Recently, I sold my MX-100 and purchased an FX-100. This printer prints twice as fast and supports proportional spacing, but the most important feature for me was the ability to download character sets. The ROM set consists of ASCII in the lower 128 positions and italics (plus a few other characters) in the upper 128. In addition, 2K RAM is provided for an alternative character set, which can be turned on and off at will. The function below loads the bit-image bytes from APLCHARS to either the upper or lower half of the alternative, RAM-based, character set. Note that I do not load the overstruck characters to the lower 128, where they have ASCII control interpretations. Such characters must be produced using the sequence of character, backspace, character. I can, however, load the overstruck characters into the upper 128 and tell the printer not to interpret them as control codes. With a function such as REVERSE (printed in an earlier column), APL characters in the lower 128 can be moved into the upper 128 and then sent to the printer. This avoids backspacing and produces faster, cleaner output. [Ed. The difference in quality between backspaced overstrucks and those printed directly is most obvious from samples sent by Steve; we regret we can't print them, since reproductions do not show the full difference.]

```
      ∇DOWNLOAD_APLCHARS[⎕]∇
[  0]    DOWNLOAD_APLCHARS ;FN;ANS
[  1]   ⍝DOWNLOADS BIT MAPPED APL CHARACTERS TO EPSON RAM
[  2]   'ENTER: FILE NAME TO DOWNLOAD TO...'
[  3]   (FN←⍞) ⎕CREATE 10
[  4]   →(0≠⍴⎕STATUS)/0
[  5]   'UPPER 128? (Y/N)'
[  6]   →('Y'=1↑ANS←⍞)/UPPER
[  7]   LOWER:⎕AV[⎕IO+27 38 0 32 127] ⎕PUT 10
[  8]     (APLCHARS[(⎕IO+31+⍳96);],⎕AV[⎕IO]) ⎕PUT 10
[  9]     →EXIT
[ 10]   UPPER:(EXPAND_PTR) ⎕PUT 10
[ 11]     ⎕AV[⎕IO+27 ¯38 0 128 255] ⎕PUT 10
[ 12]     (APLCHARS[⎕IO+¯1+⍳128;],⎕AV[⎕IO]) ⎕PUT 10
[ 13]   EXIT:'DONE'
[ 14]   ⎕UNTIE 10
      ∇ROM_CG[⎕]∇
[  0]    R ← ROM_CG
[  1]   ⍝SELECTS RESIDENT CHARACTER SET
[  2]   R←⎕AV[⎕IO+27 37 0 0]
      ∇RAM_CG[⎕]∇
[  0]    R ← RAM_CG
[  1]   ⍝SELECTS DOWNLOADED CHARACTER SET IN EPSON'S RAM
[  2]   R←⎕AV[⎕IO+27 37 1 0]
```

The upload function can be used to download characters to the printer by responding with a filename such as 'IEEE4'. You can, however, also send these characters to a disk file, such as 'epson.aplchars,' and then use the copy facilities of either mED or PIP to set up the printer without first loading the APL interpreter. To do this, you copy the file to the printer (with the printer turned on). For example, from command level of the microEDITOR, issue the command: copy 'epson.aplchars' to 'ieee4' in order to set up the printer. This is particularly helpful when using the SPET as an APL terminal. Reg Beck has provided a machine language program to set up the printer with a similar APL character set from main menu. That program is also on disk.

Finally, the Epson printer needs an IEEE488 interface of some sort. I use the board provided by Epson that fits inside the printer. This works fine but does not provide any translations for the 6502 side of the SPET. Reg Beck employs an ADA1800 interface with good results in both the 6809 and 6502 modes.

TABLE 1 -- Sample of the Zeller APL Character Set, with Column Definitions

6	inst	I_N	137	137	249	137	160	9	5	3	32	1
7	eeol	E_L	249	169	169	137	160	2	2	2	2	1
8	crfwd	C_F	113	137	137	137	81	32	21	21	17	1
9	crbck	C_B	113	137	137	137	81	32	22	22	11	1
10	tab	T_B	129	129	249	129	129	32	22	22	11	1
11	crdwn	C_D	113	137	137	137	81	32	18	18	15	1
12	crup	C_U	113	137	137	137	81	31	2	2	31	1
13	clear	C_L	113	137	137	137	81	32	2	2	2	2
14	cr	C_R	113	137	137	137	81	32	21	23	22	9
15	nor	⍱	81	137	133	131	66	35	37	41	81	1
16	nand	⍲	66	131	133	137	81	41	37	35	66	1
17	del stile	⍒	33	49	41	37	256	37	41	49	33	1
18	delta stile	⍋	5	13	21	37	256	37	21	13	5	1
19	circle stile	⌽	25	37	67	67	256	67	67	37	25	1
20	circle slope	⍉	129	89	37	83	75	71	67	38	25	1

◇◇

APL CHARACTER SET FOR FX80 The material below comes from Reginald Beck of Box 16, Glen Drive, Fox Mountain, RR #2, Williams Lake, B.C. V2G 2P2. He uses the ADA 1800 interface, which can be switched to an 8-bit mode. You must be able to transmit 8 bits to your printer (normal ASCII is sent in 7 bits). We think most interfaces will handle this, but you'll have to test. As did Steve Zeller, Reg designed his own character set--but in italics. We show a sample at the left; the reproduction doesn't show the quality of the original. Reg dumps the set to his printer before a session with APL. Here is his description of what he does and how he does it:

```
 ∇DUMP[⎕]∇
 [  0]     DUMP ;VECTOR;⎕IO
 [  1]    'IEEE4' ⎕CREATE 1+⎕IO←0
 [  2]    VECTOR←⎕AV[SWAP,ALF,ALFSYM,CHARS]
 [  3]    VECTOR ⎕PUT 1
 [  4]    ⎕UNTIE 1
```

"First, you send some bytes to direct the printer to swap its ROM characters into RAM. Then you send bytes to tell the printer where each character is to be loaded. The positions are from 0 to 255. If you send a sequence of characters, you only have to tell the printer the starting and ending positions in the 0-255 sequence; otherwise, you specify the position of each character you send. If you can use some of the ROM characters (I used the numbers and the punctuation marks), you don't have to download them as they are already in RAM. Having loaded the set you want, you then tell the printer to use the RAM character set instead of the one in ROM.

"The programs I send on disk will do this for the FX80. One APL function dumps the character set to disk (DUMP). The bytes are stored in four global vectors in the workspace: letters (ALF), the shifted APL symbols (ALFSYM), the other characters (CHARS), and 5 swap bytes which swap all printer ROM into RAM (SWAP). I concatenate these vectors into one in DUMP (see above). It's a good idea to keep separate global vectors so you can locate and change a particular character. On disk, you'll find two workspaces: 1) APLCHRSET includes a function DESCRIBE; it comes in running and supplies some information before it actually dumps the set; 2) the other function, APLCHR, automatically dumps the characters as soon as you load it. Use either one.

"Since the characters use bytes up to 256, you must use a full 8-bit interface. I show at left the steps you follow to load and use my character set."

1. Set interface to 8-bit mode.
2. Load APL (V 1.1)
3. Type)LOAD APLCHR <RETURN>
4. Listen for printer's "burp."
5. Load Jim Swift's SDUMP and dump the screen any time.

Readers fill find SDUMP on page 109 (issue 8) of the Gazette, and on the disk we offer below. After the above came in, Reg wrote an assembly-language program which loads his APL character set from main menu, filenamed: typeapl:men; it is also on disk. Put a disk with the character-set program 'apl.chr' and the program 'typeapl:men' in drive 1 (it's handy on the language disk). Type: typeapl:men <RETURN> at main menu, and the FX-80 character set will be loaded from menu--if you remember to turn on your printer....

Reg also sent an assembly-language program named 'adump,' which will send any SEQ file in SuperPET to an ieee4 printer from main menu. It can be re-assembled and re-linked for 'printer' or 'serial' printers. Very handy, in combination with another program we got from P.J. Rovero, which gets a two-column directory of either drive from main menu. That's on disk also, as: dir:men. Since Rovero's directory program loads in user memory, and Reg's dump loads in bank 15, you can use the two in tandem: get a directory, then print any file. Neat. In all languages but APL and COBOL, you can 'reset' to the language or facility which is in the upper 64 after using 'dir:men' and 'adump.' We also put on disk Jim Swift's 'loader:au,' which alphabetizes the directory of any disk (pretty fast) and then loads the program/WS that you choose. If you use SDUMP (also on disk) with it, you can send to printer an alphabetized directory of any disk from APL. Source files (.asm and .cmd) are on disk, with the .mod files, ready to run.

* * *

We also put on disk from R. D. Connely, 424 South Florida Ave., Joplin, MO 64801 a character set for BASE 2 printers (2K for U16 & 17) and a 4K version for late models (the set includes APL). You may obtain on disk copies of all APL character set articles of this issue, plus workspaces, character sets, and listings used by Peterson, Zeller and Beck. For 4040 format, write Secretary ISPUG, 4782 Boston Post Road, Pelham, N.Y. 10803. For 8050, write the Editor, PO Box 411, Hatteras, N.C. 27943. Enclose $10.00 U.S. State format. Make checks to ISPUG.

◇◇

Prices, back copies, Vol. 1 (Postpaid), $ U.S.

No. 1: not available	No. 4: $1.25	No. 7: $2.50	No. 10: $2.50
No. 2: $1.25	No. 5: $1.25	No. 8: $2.50	No. 11: $3.50
No. 3: $1.25	No. 6: $3.75	No. 9: $2.75	No. 12: $3.50

Send check to the Editor, PO Box 411, Hatteras, N.C. 27943. Add 30% to prices above to cover 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.
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: Paul V. Skipski, Secretary, 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. Secretary, ISPUG: Paul V. Skipski. Editor, SuperPET Gazette, Dick Barnes. Send membership applications/dues to the attention of Mr. Skipski; newsletter material to the attention of Dick Barnes, Editor. SuperPET is a trademark of Commodore Business Machines, Inc.; WordPro a trademark of Professional Software, Inc. Contents of this issue copyrighted by ISPUG, 1984, except as otherwise shown; excerpts may be reprinted for review or information if the source is quoted. Members of ISPUG are authorized to copy the material; TPUG may copy and reprint any material so long as the source is quoted. If you send inquiries, enclose a self-addressed, postpaid envelope (4 x 9.5 inches, please). If you submit material for the Gazette, enclose a suitable return/reply envelope, postpaid. Canadians: enclose Canadian dimes for postage. See enclosed application form for membership dues. The Gazette comes with membership.

For all outside the U.S.: All nations members of the Postal Union offer certificates good in the postage of any other country for a small charge. The Union includes most nations of the world.

FIRST CLASS MAIL

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

First-Class Mail
in U.S. and Canada