SuperPET Gazette

Just before this issue went to press, we received a letter from the Toronto Pet Users Group (TPUG) which announced the successful completion of the project to make the OS/9 operating system available on SuperPET. Manuals are ready for distribution to those who ordered; the OS/9 boards for both 2- and 3-board SuperPET models should be available when you read this.

We don't know whether or not TPUG will accept new orders for OS/9, so don't try to order until TPUG announces what it plans to do. The cost of OS/9 (for manuals and hardware) has been held to $149 for those who made a deposit of $68.09 some months ago. For those who came in late: OS/9 is a UNIX-like system which now runs on the Radio Shack Color Computer, and for which a substantial amount of software is available (see I, No. 14, p. 261; and I, No. 15, p. 291). We hope to have a fuller report on OS/9 as implemented in SuperPET in the next issue.

ON BEDIT, CLM, GSCROLL BATCH FILES and SUCH In recent issues, we've mentioned a new and markedly improved Editor (Joe Bostic's BEDIT), a new operating system overlay by John Toebes (called CLM, or Command Line Monitor), a much-improved GSCROLL by Terry Peterson (which patches the ROM bug on interrupt routines in the banks and puts instant phrases on the keypad for the hunt-and-peck typist, among other virtues), plus Loch Rose's gem, COPY/KILL, which makes copying files from disk to disk a pleasure, and Alain Proulx's DOS, an ML program which handles all DOS commands from menu in two versions--one in English, the second in French. CLM, GSCROLL, and BEDIT all implement BATCH files and let SPET execute them whilst you sip a martini or have a nap. We hoped to announce that all this stuff was available on an ISPUG utility disk this issue, but ran into some problems and delays:

First, we're trying to marry a program called CALC (which makes a super-calculator of SuperPET) with BEDIT, or at least an optional version of BEDIT, so all of us can cease the nonsense of doing our math and hex/decimal/binary conversions on a calculator (after copying from the screen--with errors) while our expensive computer sits there stupidly unable to do what a $9.95 Sears special can do.

John Toebes (author of CLM) is ill and in the hospital; his SPET expired, smoking, from a software crash (!) and hasn't been fixed, so CLM is not finished. It took longer than expected to eradicate the last bugs from BEDIT, which is now in limited distribution for final test before release. Terry Peterson has married CLM and GSCROLL most successfully, but the new version has not yet been extensively tested. We won't issue software untested; it disappoints the user and becomes a nightmare to update and fix. Last, the new programs are not yet properly documented. We'll do our best to release the disk by next issue; meanwhile, please don't order it. We will say the disk will be worth waiting for.

DO YOU SEE RED WHEN YOUR ADDRESS LABEL IS READ?

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 renew early, before our heartless program casts you from our disk files into the land of the expired.

ONCE OVER LIGHTLY Miscellany We've said this before, but apparently must say it again, because letters of protest still arrive. Our mail labels, bearing notices saying, "Memmbership EXPIRED" are made up about a month before <u>you</u> get them on a copy of the <u>Gazette</u>. If, meanwhile, you have blessed us with a remittance, you still get the notice. Last issue, we sent the labels and copy to the printer (who mailed the issue) on Dec. 26; you probably got the issue in late January. Please don't burden ye ed and the noble folks in the Postal Service with an indignant letter. Just ask yourself if you paid up about the 1st of the month on which the issue arrived. If so, the labels were printed before your check arrived. Until we own our own print shop, get the Postal Service to deliver mail an hour after deposit, and print mail labels just as the Gazette drops into the mail slot, concurrent we and thee ain't.

MORE ON MUMPS Seems Jerry Carroll of Woodland Hills, CA, had trouble sending MUMPS output to printer. Dan Jeffers of Honolulu sends the tips following: In MUMPS, device 2 is labelled printer; Dan thinks this device uses the IEEE-488 and translates everything to PET ASCII. Device 3 is for the serial port, and it works (you must have positive voltage on pins 5, 6, and 8); if you have a parallel printer, you're out of luck without an interface from serial or IEEE to parallel. It's possible to drive a parallel printer from the user port, but no software to do it is in ROM, either in 6809 or 6502. You must write your own. Dan adds there's another option: use Device 7; it will access any device you can reach from the Waterloo interpreters. Dan uses it for his IEEE printer, whose address is set to device 5. In MUMPS, he connects the printer to the IEEE bus and issues this command: OPEN 7:"W,IEEE5". This works and bypasses the unwanted translation to PET ASCII. Dan also says that he talked to David Brown, who converted MUMPS to SuperPET, and understands that REL files on 8250 and 9060 hard drives are substantially bigger than 720 sectors, so that there is more space on the drives for MUMPS global files than on other drives.

PASCAL COMPILER Daniel Wiedman of 1541 Swallow Drive, Brentwood, MO 63144, says he uses the Oxford Pascal compiler on his C64 in conjunction with SPET, editing and debugging on SPET and then compiling and running on the C64. He says the Oxford Compiler accepts ASCII files, which he transfers between computers with an RTC Link II on the C64. For more details, write, or call 314 968 9672. See also Bob Davis' article on a Pascal compiler which runs on SPET's 8032 side, in II,7.

OUR GURUS EXPLAIN THE WIZARDRY Last issue, we printed reports from Loch Rose and Frank Brewster of weird happenings to printers on the IEEE-488 and on the serial port. Frank complained that he sometimes crashed when he turned off his printer on the serial port. Gurus Terry Peterson and John Toebes report that the 6551 ACIA sometimes freezes up with its IRQ asserted and that some 6551s will issue IRQs caused by RS-232 noise, even when the 6551 is supposed to be disabled. This may explain Frank's problem; it may also explain why he crashes once in a while when he turns his modem off. Ameliorative: do not turn off printer or modem during a session, once they're on.

Loch Rose's printer output a DOS command when he turned it on after using his disks and computer for a while and then sent a DOS command to disk. Terry Peterson reports that Loch's ADA printer interface powers up "listening" and stays in listen mode until it receives an "unlisten" command at the end of the command sent to disk. Terry reports the same thing happens with his VE-2 interface from AB Computers. Cure: put a blank line to printer or send a command to a non-existent IEEE device. Either action will "unlisten" the printer without printing a character. Guru Toebes assigns the same causes and prescribes the same cures, but blames bad code in our ROMs; the code should "unlisten" a newly-turned on device unless a command is directed to it.

Loch also complained that his printer slowed to a crawl if his disk drives were off. Terry reports that the problem is caused by the dead weight of the disk drives on the IEEE bus; the IEEE specs call for more than half of the connected devices to be powered. By our count, Loch had half his devices off, hmmm?

We're not inclined to believe in magic, but only the black variety explains the remaining puzzle. We've had two calls from the West Coast from people trying to hook printers to the serial port. Neither could get a linefeed with a CR until he turned on the "linefeed with CR" switch on his printer. Yet the code sent by the "put serial" routine in ROM explicitly sends a linefeed with each CR (trace the code starting at $C7C1, as Terry did). Most of us get a linefeed with each CR on the serial port. Two people don't. Toadstools and gravedust....

OBSCENITY In an age which permits topless waitresses and sex on screen, we find it strange indeed that 1) fasteners (nuts, bolts, screws) and 2) ON/OFF switches have been declared obscene (why else are they hidden from view?). Why must we fumble around the backside (blush) of our computer to lay our hand on the utterly unmentionable device which turns the computer on? Back in the good old days of chaste maidens, ON/OFF switches and fasteners were right out there in the open where you could (blush) fondle 'em any time. If your disk drives are on a shelf under your computer desk, you must hire an octopus to reach the power (censored). Being tired of taking off our girdle, using mirrors, and standing on our head to find things, we wish the makers would end their Victorian prudery on switchery and screwery.

GAY COMPUTER NETWORK The Gays News Information and Communication Network sent a flyer announcing itself, and said: "...the only way for new members to be comfortable with the service...is to get hands-on experience." That's news?

SOME 4040 DOS PROBLEMS Paul Matzke of Madison, WI, has run into problems using REL files with his 4040 drive: 1) He says his 4040 forgets how to scratch a file in mBASIC after access to a REL file until he reads a SEQ file or lists the directory of the other drive. A command to scratch file "foo" fails immediately after access to "foo,rel". Guru Terry Peterson says that mBASIC seems to remember that "foo" is a relative file even though it has been closed. A command to scratch "foo" keeps mBASIC from searching for it as a SEQ file (scratch commands don't take DOS format designations) and mBASIC knows it is a REL file. Next, 2) Paul must get a directory from the drive not containing a relative file to avoid closing the file. Terry reports that this is an old 4040 bug; calling up a directory on either drive closes any open files on that drive. Neither 8x50 nor 1541 drives with DOS 2.7 close files at a directory call; status of the 2031 is unknown, as is that of pre-DOS 2.7 8050 drives.

ADDRESSES, NOT STRINGS, ARE SWITCHED In Issue 15, Vol. I, p. 270, we noted that, for some strange reason, a shell sort ran faster when we switched strings in microBASIC than when we switched subscripts. In Microsoft BASIC, subscript switching is always faster. Now, John Toebes tells us why: Waterloo designed mBASIC so that when you transfer strings the language merely switches their addresses, not the strings themselves. It's a splendid arrangement. Too bad that archaic, unstructured Microsoft BASIC still comes with most new computers. The vendors argue it makes their machines compatible with the old software base. Gee, why don't new cars arrive complete with whip, harness, and horse--to be compatible with the old, free, green fodder base all around us? Take note, GM.

HOW TO BECOME AN AUTHOR A lot of folks write but few become authors; in our lexicon, a writer turns author when published. Now, to accompany instant orange juice, you may become an instant author. We got a blurb from a software house on an "authoring" language for school courseware. Golly, think of the wide-open opportunity! Be first to market with a "doctoring" language. Instant M.D.! Then try mayoring, detectiving, teachering and even thiefing. Or should people who use "authoring" examine the legitimacy of their fathering and mothering?

STARVATION DIET, THIN GAZETTE
an Editorial

I come out from behind the editorial "we" and speak plainly: I'm damn tired of writing most of every Gazette for free. The number of contributed articles has dropped off to practically nothing in the past few months. Were it not for the work of our Associate Editors--likewise unpaid--I'd have given up this job long ago. Starting next issue, I've set a quota on what I'll write. If you start getting thin Gazettes, you'll know why--you've failed to meet the ultimate test of a user group; you're reading but not writing, taking but not giving. The purpose of ISPUG is the interchange of useful information between its members.

Of late, the interchange has become a one-way street. I have no motive to continue editing and publishing the Gazette under such circumstances. The Gazette needs articles, notes, and intelligent comments--not programs, of which we have a bushel; most are much too long to publish and useless without explanation.

In sum, start giving or you're not going to get. --Dick Barnes

---

A REPAIR TRIP TO THE COMMODORE MAIN PLANT

by Tony Klinkert
Box 110996
Carrollton, TX 75011

This past December, I brought my SuperPET system in for maintenance at Commodore's plant in West Chester, Pennsylvania, and turned the event into an opportunity to investigate and report to the SPET community about Commodore's service network in general and repair support at this plant site in particular. I've subscribed to the Gazette since its inception, and hope this article helps other readers, in return for the assistance other contributors have provided to me.

My system problems began when my Micropolis 8050 drive acted strangely; drive 0 would "run" even without a diskette, although when I put a disk in it, it worked as usual. Then my 8300P printer went awry. The ready light would not come on; whenever the printer was on the IEEE-488 bus the system would not operate; when the printer was disconnected, the rest of the system was fine. It was time to take the system in for repair.

I purchased my SPET in May, 1982. Since I was taking in my printer and drive, I thought I'd have whoever did the work look at SPET as well, clean the keyboard and install the PaperClip ROM I had just purchased. But where would I go for repairs? The failures occurred while I was training in the West Chester area for my next assignment in West Germany. Before coming here, I obtained service at Eclectic Systems in Dallas, where I purchased my system. Rather than look for a dealer here, I decided to go straight to the Commodore plant.

My first call to the plant was to Mr. Akbar Teymourzddeh ("Teymour" for short), Senior Service Technician for the CBM Customer Service Department. He is a likeable fellow who will spend a few minutes (though he seems very busy) talking to customers about their repair problems. After going over mine, he invited me to bring in my system. Teymour told me this this plant makes and repairs all computers in Commodore's line-up. His department is responsible for a large volume of repair work coming mostly from dealers and from individuals--via UPS, the mail, and on a "walk-in" basis (walk-ins are permitted from 10 a.m. to 12m, Monday through Friday).

From Teymour, I also learned that Commodore is negotiating with RCA, Honeywell, and Western Union for a national repair network for its consumer products line (C-64, Vic, etc.). Commodore's business machines--and SPET--are serviced only by dealers and at this plant. Though he has not heard of SPET going out of production, he believes that Commodore will continue to support it at this site, in production or not.

So, with a Chester County map in hand I set out for the plant. From Philadelphia I drove about 20 miles West on Highway 76, turned south onto Highway 202 at the King of Prussia exit, and proceeded 17 miles to the Paoli Pike exit; there I turned left (East), proceeded one mile, and turned left again onto Ellis Road, (between the Church and the Amoco station). One mile up, I turned left onto Wilson Drive and into the Brandywine Industrial Park. The plant lies at the end of this road.

There, Teymour met me and told me that repair costs would be $50 per hour, guaranteed for thirty days. He gave me a direct number for the shop in case I had any questions while my system was there (the normal service number is 215-431-9106). He took the system and told me to call back in one week, which I did--and discovered what my problems were. The disk drive had a bad IC ($50 labor and $2.50 parts); my printer had a bad microprocessor ($50); it cost $50 more for the work to install my ROM and to give SPET a complete checkout. As a freebie, Teymour threw in a spare print wheel, a disk drive reference manual, and factory boxes suitable for shipping the system to West Germany. Then I bought the SPET and 8300P technical service manuals. My total bill was $202.50. Considering the work done and the assistance, I feel it was worth it, and so wrote this article so others would know about the services Commodore makes available.

On my way home, I stopped by a Federal Express office to weigh the boxes and have them quote a price for shipment from anywhere in the domestic US to West Chester. UPS later gave me their prices. The figures appear below. Data for the 8300P is missing because the volume exceeds the maximum for both Federal and UPS. (Ed. Xerox will service the 8300 at any service location. The figures below are high for short distances. We shipped SPET 300 miles by UPS, insured, for $10.51, with delivery on the next day. It came back in fine shape.]

Box Contents	Weight	Dimensions (inches)	FedEx Next Day	FedEx Overnight	UPS Regular	2nd Day	Next Day
8050 drive	33 lbs.	20x20x11	$44.50	$72	$15.05	$34.00	$46.00
SPET	46 lbs.	24x21x19	$56.50	$85	$20.46	$46.50	$60.00

ON MICROSPACING PRINTERS If you'd like to draw highly accurate graphs with your printer, or to write a printer formatter to proportional-space justify text (put the same number of spaces between words and produce a flush-right margin), you can do it if your printer will microspace.

Gee, what's a "microspace?" On our printer, it's a carriage movement of 1/120th of an inch horizontally, which seems to be fairly common. Vertically, it is 1/48th of an inch (the paper moves up or down that distance). On all printers which can microspace, you usually find ESCAPE sequences which will 1) change the default settings for both horizontal carriage motion (the Horizontal Motion Index, or HMI) and 2) the Vertical Motion Index, or VMI. Printers usually default to one or another of the following HMI values:

HMI for 10-pitch (10 characters/inch, horizontally): 12/120ths or 1/10th of an inch per character space.

HMI for 12-pitch (12 characters/inch, horizontally), which you now read: 10/120ths or 1/12th of an inch per character.

HMI for 15-pitch (15 characters/inch): 8/120ths, or 1/15th of an inch.

Well, if we can change from a printhead movement, per character, of 1/10th of an inch to a movement of 1/15th of an inch, our printer must move in an increment which is common to all values, which very obviously is 1/60th of an inch. If your printer will output 10 through 15 pitch, it should microspace at a minimum increment of no more than 1/60th of an inch, and perhaps at 1/120th.

Somewhere, your printer manual should tell you how to change the HMI for the various sizes of type, and, if you're lucky, how to set the HMI so you can get a minimum spacing of 1/60th of an inch or less, horizontally.

Similarly, most printers will let you change the number of lines you print per vertical inch. The common value is six such lines, for both 10- and 12-pitch type. The text you now read is spaced at six lines per inch. Many printer will give you increments of 1/48th of an inch for vertical paper motion (VMI). If you check that value out, you'll see that normal six-line spacing is 8 vertical microspaces per line (48/8=6). Again, if you're lucky, you'll find the command sequence which will set to a minimum increment of 1/48th of an inch for each linefeed or negative linefeed sent to printer.

We show below the ESCAPE sequences which set DIABLO's HMI to 1/120th of an inch (domicro$) and then return it to normal 12-pitch character spacing, as an example. The strings can be sent to the printer at any time; they are parameter commands and print absolutely nothing. We'll demonstrate how to use them to proportional-space text and justify it, and to make very accurate graphs.

```
domicro$=chr$(27)+chr$(31)+chr$(2)
donorm$=chr$(27)+chr$(31)+chr$(11)
```

For reasons we'll not explore in detail, we had to write a new printer formatter to publish the Gazette. No word-processing program we've seen (or dedicated word processor, either) is any damn good when you must print long documents. When we wrote our formatter, we included proportional-spacing between words so we could right-justify. This text is so printed, using microspacing. Here's how:

First, input whole lines from disk. Find 1) the line length and then 2) count the spaces in the line (intrinsic function "idx" does this splendidly). Remember that blanks following the end-of-line are not spaces, but nulls. If we assume 1/120th of an inch microspacing, print in 12-pitch, and justify only lines which

```
if len(line$)>72 and len(line$)<80
   microspaces%=800-(len(line$)*10)
   micro_per_space%=microspaces%/spaces%
   odd_spaces%=mod(microspaces%,spaces%)
endif
```

are longer than 72 characters, the program at left will figure out how many microspaces we must place between each word if we're to build a justified right-hand margin. Since the approach may not be clear, let us dissect it a bit. How many increments of 1/120th of an inch are found in a 12-pitch line of 80 characters in length? Each character takes up 10/120ths of an inch, so 80 characters must then occupy 80 times 10/120ths, or 800/120ths inches. Now, suppose we have a line of 75 characters and 11 spaces. The first line above says 800-(75*10), or that we need 50 microspaces (five regular spaces) to fill the line flush right. The second line tells us that micro_per_space% (how many microspaces we must stuff in between each word) equals 50/11 (note that we use integers so that this becomes a value of 4). But what do we do about the remainder of the microspaces, which won't integer divide? Good old intrinsic function "mod" gives us that number of spaces directly, for mod(50,11) yields a value of six.

If we stuff between each word, in addition to a normal space, four microspaces per space, we get a total of 44. If we add in one of the six remaining spaces between each word until we run out, we get our total of 50 microspaces.

Suppose we parse the string so we print a word at a time, including its suffixed space. It's then simple to stuff in the microspaces. Pretend we have a variable "toggle%", which tells us that we have a line which needs microspacing. Then:

```
if toggle%
  if odd_spaces%                                          ! Suppress all CR's with
     add%=1 : odd_spaces%=odd_spaces%-1                   ! semicolons.
  endif
  print #25, domicro$;rpt$(" ",micro_per_space%+add%);    ! Microspace.
  print #25, donorm$;                                     ! Resume normal spacing.
  add%=0
endif
```

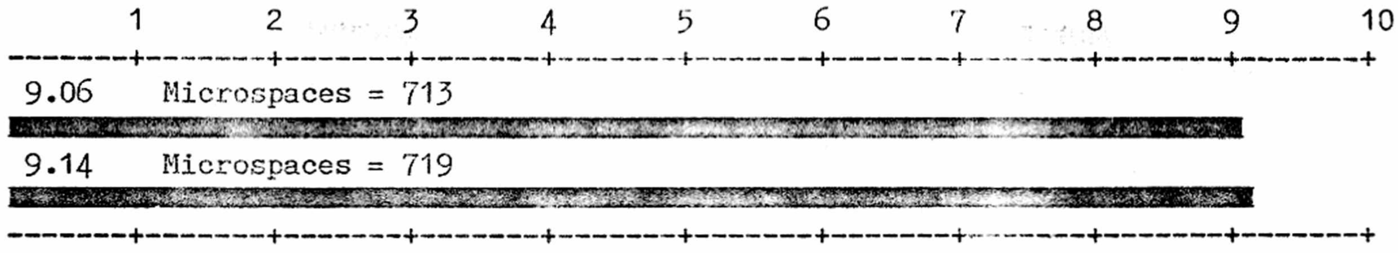
We print the microspaces after the normal one from the program above. About half the lines on this page hold an odd number of spaces; do you see the extra microspace between the first few words, as compared to the spaces toward the end of the line? We can't. We now pass on to the subject of graphing with microspaces.

BULLFEATHERS vs. BIRDSEED
Graphs from Microspacing

Having peered into how to justify text with microspaces, let us take a look at what can be done to draw precision graphs with a daisy-wheeler. Any good dot-matrix printer should be capable of even better work. Don't be conned into the belief that we don't need microspacing at 1/120th or 1/60th of an inch to draw suitable graphs, as we'll demonstrate below.

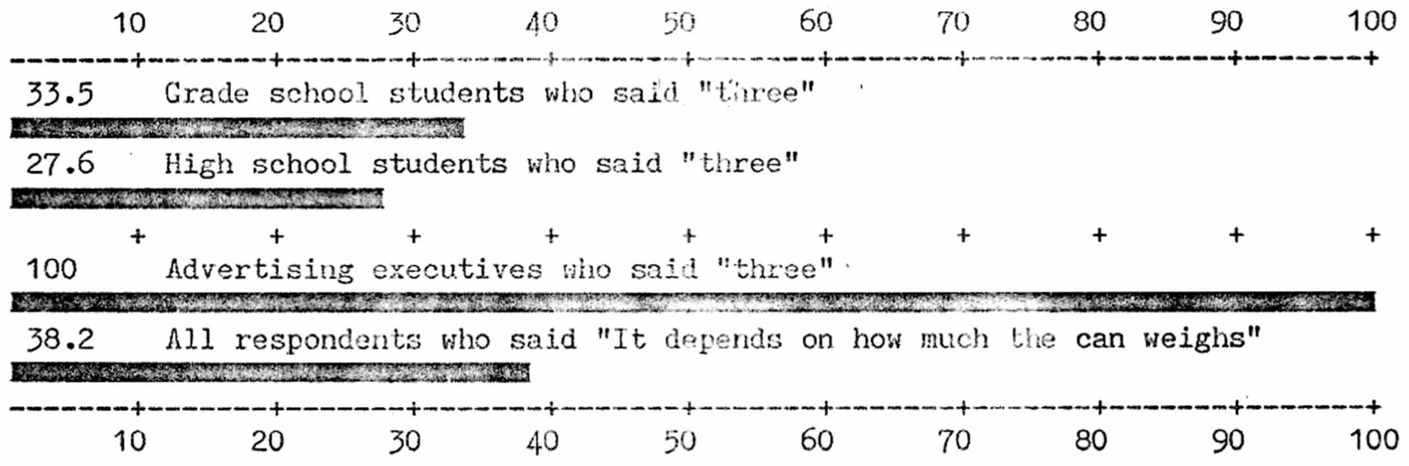
Neither the screen of SuperPET nor any printer set to normal character spacing will draw any graph in increments less than one full character space--which is often grossly out of scale. Daisy_wheel printers offer no "graphing" character which will form solid bars--unless you microspace and overprint. No printers will graph values accurately unless microspaced: try to draw the difference between 9.06 and 9.14 without microspacing; then see the graph below--microspaced.

COMPARISON OF BARS PRINTED WITH FRACTIONAL VALUES OF 9.06 and 9.14



Let's examine the problems we face when we draw. We'll graph the responses to a question: "If one can of birdseed weighs three times as much as a can of bull-feathers, how many pounds of bullfeathers will equal a pound of birdseed?" We graph the responses below:

RESPONSES TO BULLFEATHERS vs. BIRDSEED (in Per Cent of Respondents)



The originals of these graphs show good, solid black bars; unfortunately, they will reproduce with gray in the center of the bars.

First problem: how do you form solid bars on a daisy-wheeler? If you print a vertical bar "¦", it starts in the <u>middle</u> of a character space at left margin, no matter how fine the microspacing. The bar is indented; your graphs do not start at zero. You must pick the widest character you can print, and one with vertical sides. The capital letter "M" is not ideal, but serves. Overstruck at fine microspacing, it forms a bar which starts at zero and is square at the end. A problem occurs at very small bar values--you can't overstrike enough times to form a bar to scale. So, leave out the bar and substitute a note about the numeric value. Last, you find that the right half of "M" will stick out beyond your scaled value (if you're accurate to 1/120th of an inch in scaling, the character you print should be 1/120th of an inch wide, right?). You must, therefore, subtract some microspaces from the width of "M" to end the bar on the right value.

The second serious problem is scale. You certainly don't want to have a scale in increments of 1000 when your largest bar value is in 80. We found a simple way to vary the scale by four orders of magnitude (1...10, 1000...10000) automatically, depending upon the maximum bar value; it also accepts decimal fractions, and prints them, rounded, to the nearest 1/120th of an inch.

The third problem is page width--how do you divide 6 5/8 inches of page into 10 even increments (ho! for the arrival of metrics). If you use a normal 12-pitch character spacing when not microspacing on an 80-character line, an increment of eight spaces between scale marks turns the trick (see above). You then have 800 microspaces of 1/120th of an inch for a full page width, or 80 microspaces for each major scale increment (in the sample above, we have 80 microspaces for each scale increment of 10). Only the bars are microspaced, and, for simplicity, they are printed at a steady microspace of 1/120th of an inch. It'd be far faster to microspace at 1/60th or less almost to the end of the bar, and then set to the finest increment possible--but it takes more program (and space we don't have).

The program below has been cut to bare bones (no error or input traps, no bells or whistles) simply to demonstrate how easy graphing is on a daisy-wheeler. It should be equally simple on a dot-matrix printer which will microspace to 1/60th of an inch. With a little imagination, dot-matrix owners should be able to print some very fancy graphs. Both graphs in this article were printed with it; you may plot as many bars as you wish; for each two bars the scale marks "+" are repeated; on long graphs, the number scale is repeated at the bottom. We used the format for bar graphs developed by Delton B. Richardson of Norcross, Georgia.

We add that any printer which can't do a negative linefeed or backspace ought to be tied to the legs of its designer as he is thrown overboard. Such a beast can transform graphics plots and printout of two-column text into a nightmare. Avoid such printers as you'd avoid an Edsel (if you know what <u>that</u> was, you're getting old!).

```
100 ! graph:bd. A simple bar graph program with microspacing.
110 !                     DEFINE CONSTANTS
120 domicro$=chr$(27)+chr$(31)+chr$(2) : D$=chr$(10) : CR$=chr$(13)
130 donorm$=chr$(27)+chr$(31)+chr$(11) : CS$=chr$(12): L$=chr$(8) ! Backspace
140 !                      GET BAR DATA
150 print CS$;
160 print "Enter the title of the graph in one line or less."
170 input "", title$
180 half%=(80-len(title$))/2  : title$=rpt$(" ",half%)+title$  ! Center title
190 print : input "How many bars will you draw? ", num_bars%
200 print
210 for i%=1 to num_bars%
220   print "What is the value for bar"; i%;"? "; : input "", bar(i%)
230   print "What is the name of the bar? " : input "", bar_name$(i%)
240   print
250 next i%
260 !                DETERMINE THE MAXIMUM BAR VALUE
270 max_value=bar(1)
280 for i%=2 to num_bars%
290   if bar(i%)>max_value then max_value=bar(i%)
300 next i%
310 !            DETERMINE SCALE FOR BARS AND PRINT THEM
320 if max_value<=10
330   scale%=10 : long%=2         ! This is a simple-minded approach to save
340 elseif max_value<=100         ! space, limited to a maximum scale of 10,000
350   scale%=100 : long%=3
360 elseif max_value<=1000        ! Larger values can easily be handled by
370   scale%=1000 : long%=4       ! showing scale x factors of 10 after the
380 elseif max_value<=10000       ! bar name, as in: (in thousands)
390   scale%=10000 : long%=5
```

```
400 endif
410 open #12, "ieee4", output
420 print #12, title$;D$ : call scale                ! Print title, scale,
430 call border(1)                                    ! and top border.
440 for i%=1 to num_bars%                             ! Set no. of microspaced
450   num_strikes%=(800*(bar(i%)/scale%)-12)+.5       ! strikes, name bar, set
460   print #12, bar(i%);tab (10);bar_name$(i%)       ! microspacing, print bar,
470   if num_strikes%>=8                              ! cancel microspacing.
480     print #12, domicro$; rpt$("M",num_strikes%); donorm$
490   else
500     print #12, "(Bar too small to print to scale)"
510   endif
520   if not fp(i%/2) and i%<num_bars% then call border(0) ! Insert scale
530 next i%                                            ! between each two lines.
540 call border(1) : if num_bars%=>4 then call scale ! Repeat scale at bottom
550 stop                                               ! on long graphs.
560
570 proc scale                                         ! Print scale values.
580   for i%=scale%/10 to scale% step scale%/10
590     print #12, rpt$(" ",9-long%);value$(i%);
600   next i%
610   print #12
620 endproc
630
640 proc border(all)
650   if all then print #12, rpt$("-",80);rpt$(L$,80);   ! Border & backspace.
660   for j%=scale%/10 to scale% step scale%/10
670     print #12, rpt$(" ",7);"+";                     ! Print divisions.
680   next j%
690   print #12
700 endproc
```

We'll cover two-dimensional graphs, with ordinate and abcissa, next issue.

UNDOCUMENTED SYSTEM ROUTINES Part II

by John A. Toebes, VIII

This is the second in a series by John on the system routines in SuperPET's ROMs. The Jump Table address is the actual starting address of the routine. If John CALLs a system routine, he assumes the use of CALL MACRO (I,158) for passing parms and clearing the stack. The term "FCB" means File Control Block, which is returned by the system each time you open a file. We hope to dissect this block in a separate article.

___MOD : WSL support routine for modulus : at $B069; Jump Table $B6D8

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

The result returned is the remainder after dividing P2 by P1; it is negative if both operands are of opposite signs. 10 MOD 3 returns 1; 10 MOD -3 returns -1. This routine uses the same routines as ___DIV but takes the result from a different location.

```
Example - LDD   #20
          PSHS  D
          LDD   #6
          JSR   ___MOD ;D now has 2 in it (remainder of 20/6)
          ...   ;no LEAS is required to remove the 20
```

_RSHIFT : WSL support routine for right shift : at $B06C; Jump Table $B748

P1 - Signed Number of bits to shift right : P2 - Integer value to shift

Result - P2 shifted right P1 bits with P2 removed from the stack

This routine is used by the WSL languages to perform a right shift on a 16-bit unsigned integer. Bits shifted out on the right are lost while zero bits are shifted in on the left. If the number of bits to shift is given a minus sign, it will shift left. This routine optimizes all cases, although the code isn't very good, so that a shift right of 0 produces the input value and any shift of more than 15 produces a zero value. Cleans P2 from the stack.

This routine calls the left shift routine. Although it makes it more flexible, the overhead outweighs its usefulness. For values of 1 or 2, I recommend that you hard code the shift instructions; for values larger than 2, you may construct a loop that is far more efficient than calling this routine.

```
Example - LDD    #$1230
          PSHS   D
          LDD    #3
          CALL   _RSHIFT ;D now contains $0246
          ...    ;no LEAS is needed to remove the $1230
```

_LSHIFT : WSL support routine for left shift : at $B06F; Jump Table $B74A

P1 - Signed Number of bits to shift left : P2 - Integer value to shift

Result - P2 shifted left P1 bits with P2 removed from the stack

This routine is used by the WSL languages to perform a left shift on a unsigned 16 bit number. As in _RSHIFT, bits shifted out on the left are lost while zero bits are shifted in on the right. A negative shift count causes a right shift to be performed while values greater than 16 cause a result of 0 to be returned. _RSHIFT removes the parameter P2 from the stack; you needn't clean it up.

This routine internally checks the sign of the shift count to determine if a right or left shift is to be performed. It is of dubious value to the programmer concerned about speed. As for space, a shift of less than 5 bits can be done in the same amount of code as the overhead of calling this routine.

```
Example - LDD    #$1230
          PSHS   D
          LDD    #3
          CALL   _LSHIFT ;D now contains $9180
          ...    ;no LEAS is needed to remove the $1230
```

CARRYSET_ : Routine to check the carry bit : $B072; Jump Table $B6C4

No parameters. Result - Flag indicating if carry bit is set or not.

This routine is used by the WSL languages to get a value indicating the status of the carry bit. It returns a -1 if the carry is set and a 0 if it is not.

The usefulness of this routine approaches zero for almost all levels of programming. Internally, all it does is test the carry flag and load the appropriate value. It likewise sets the zero/non-zero condition code; after calling it you can test the EQ/NE flag.

```
Example - ...
```

```
        CALL   CARRYSET_     This is of no utility; you could have tested the
        IF     NE            CARRY flag directly without calling this routine in
           JMP    WASSET     the first place.
        ENDIF
```

TIOINIT_ : Initialize the local terminal : at $B078; Jump Table $D4CC

No parameters : Does not return anything important.

This routine resets the CRT on the SuperPET to default configuration. It also sets the default tab stops, clears the screen, selects the Waterloo ASCII character set and restores the standard IRQ interrupt handler. Finally, it resets the CRT controller chip to default state. You may use this routine in any program that reconfigures the terminal, to ensure that SuperPET is left in a known state when the program ends.

Internally, this routine calls two other useful routines. The first, CRTINIT_, ($D63A) sets up all the hardware and screen memory. The other, SYSINIT1, ($DD48) sets up the interrupt vector, initializes the keyboard/screen descriptor, and empties the keyboard buffer.

```
Example -          CALL   TIOINIT_
```

TPUTCHR_ : Put a character to the local terminal : $B07B; Jump Table $D4F0

P1 - Character to output. Does not return anything predictable.

This routine outputs the single character P1 to the terminal screen. Writing a character through this routine is identical to writing a character to a file opened to TERMINAL for output. The primary difference between calling this routine and calling PUTCHAR_: PUTCHAR_ writes to the file control by a global FCB STDOUT_ ($006D). As a result, PUTCHAR_ may be deflected to another file under program control while TPUTCHR_ always writes to the terminal screen.

This routine is always faster than writing to a file opened to TERMINAL because the output routines don't have the overhead of figuring out which device to write to.

```
Example - CALL TPUTCHR_,#12   ; clear the screen
          CALL TPUTCHR_,#'H'  ; output an H
```

TGETCHR_ : Get a character from the local terminal : $B07E; Jump Table $D4D2

P1 - Address of single byte to flag EOR (a Carriage Return)
Returns - Next character input from the terminal

This routine inputs a single character from the terminal. Getting a character through this routine is identical to getting a character from the file TERMINAL opened for input. This routine differs from GETCHAR_ in that GETCHAR_ reads from a file controlled by a global FCB at STDIN_ ($006B); input from GETCHAR_ can be deflected to another file. As with reading from TERMINAL, characters typed-ahead are input.

This routine is faster than going through the standard I/O routines; it doesn't have to determine where the input comes from. A strange aspect of TGETCHR_ is in the way it indicates EOR. When a carriage return is input, this routine sets the

```
Example - CLR   EORFLAG
          LOOP
            CALL TGETCHR_,#EORFLAG
            TST EORFLAG ;got a cr yet?
          UNTIL NE
```

single byte pointed to by the parameter P1 to a 1. It does nothing to the byte when another character is input; you must manually clear the flag before you can rely on its value.

SPUTCHR_ : Write a character to the serial port : at $B096; Jump Table $D598

P1 - address of FCB opened to the serial port. P2 - character to send.
Does not return anything predictable

This routine is used internally for I/O to the serial port. It checks on the serial port before it sends a character to ensure that it is OK to send. It also checks for an I/O TIME OUT to prevent an infinite loop in waiting for the port to clear before it sends a character. It also checks the FCB to ensure that no error condition is present, and updates the FCB and the system error message if a TIME OUT occurs.

```
Example - CALL   OPENF_,#SERIAL,#WRITE
          STD    SIOFCB
          CALL   SPUTCHR_,SIOFCB,#$0d ;Send a <CR>
                 ...
          SERIAL FCS   'SERIAL'
          WRITE  FCS   'W'
          SIOFCB RMB   2
```

Internally, this routine depends upon the address of the serial port being contained in the FCB. It is used by the host communications routines to perform all of the host file transfers.

A HEAP ON THAT STACK
or
We Blow Away Some Fog

Assembly language is not black magic, but it sometimes seems to be--especially when you first try to comprehend the workings of the hardware and user stacks. We'd guess most of the confusion arises from the terminology programmers use to describe the stack--and from weirdly wrong analogies writers employ when they talk about the stack.

What analogies do we mean? There are two. In the first, folks compare the stack to a cartridge clip, from which you can remove only the last cartridge loaded; none of the others can be reached. In the second analogy, writers picture the stack as a pile of bricks and say you can act only on the top brick. Both analogies are totally false; we wish people wouldn't write such nonsense. We'll later show why.

If the analogies are bad, the words writers use to describe the parts of the stack are worse. Why? Wherever the stack is started, it stacks downward. If, for example, the stack begins at decimal 2000 in memory, any 16-bit PUSH on the stack stores values at decimal 1999 and 1998; the next PUSH at decimal 1997 and decimal 1996. And what do the writers call decimal 1996? The TOP of the stack! And where's the bottom? Why, at decimal 2000, where we started. Confusing? Utterly! Don't you always think of high user memory, at $7FFF, as being the top of user memory? Forget it when dealing with the stack. The top of the stack is the lowest location in memory you have filled with a stack value. (If you pretend the stack is sited in antipodean China, the "top" is indeed the "top!")

Let us now look at the two components of stack handling, 1) the Stack Pointer or S register, and 2) the stack itself. Beginners tend to confuse the two. What we say about the Stack Pointer or S register applies equally to the U register of the 6809, the User Stack Pointer.

The Stack Pointer is not a memory location, but a 6809 register. It points to the location in memory where the stack itself is formed. If, for example, the Stack Pointer contains 1000, the next 16-bit value PUSHed on the stack is stored at memory locations 999 and 998. How come? It's a convention that the 6809 S register (Stack Pointer) points to the last stored value. A PUSH, therefore, always fills the next lower memory locations. In distinction, a PULL always gets back the data at the position pointed to. Suppose the Stack Pointer holds 998, and that we PULL a 16-bit value. We retrieve, from the stack, whatever data is in locations 998 and 999. The Stack Pointer then points to 1000.

The U Stack Pointer works exactly the same way--except that you must give it a starting address and reserve some room for the stack itself. Unlike the S Stack Pointer, the U Stack Pointer doesn't hold an address until you give it one in your assembly language program. There's one big exception: Waterloo uses the U stack for bank-switching in the languages/facilities of SuperPET. Don't try to use the U Stack Pointer or U stack when a language or a facility is loaded, unless you safely store the values there and get 'em back when you are done.

```
6809 Stack Pointer
        U
      1000

   The Stack
Memory         Value
Location:     on Stack

1000           None <-- Stack Pointer at Start and
 999             01     after five PULLs.
 998             02
 997             03
 996             04
 995             05 <-- Stack Pointer at 995 after five 8-bit PUSHes.
```

To make this clear, let's set the U Register at 1000 and PUSH decimal digits 1-5 on the stack. The 6809 assembler will PUSH only 8 bits at a time when you PUSH values from any 8-bit register (CC, DP, A or B registers). After five PUSHes on the stack, we find the values at left. Where is the Stack Pointer when we're done? It points to the last memory location we filled--995. If we then PULL 8 bits 5 times, into either the A or B accumulators, we get back all five digits and return the U pointer to 1000. But--suppose we want to get the first value we PUSHed (at 999, above) after five PUSHes; the Stack Pointer then points to 995. How do we get "1" at memory location 999? By the stack-of-bricks/cartridge-clip analogies, we can't do it. Malarkey. We simply write "ldb 4,u"; that means "Add 4 to the address now in the U pointer; load B register with the value at that address." The U Stack Pointer holds 995; add 4, and you point at location 999. You load your "1" in B register. So much for all the arrant nonsense about stacks of bricks and cartridge clips. You can load anything on the stack at any time--just as you can from any other set of memory locations in SuperPET.

Obviously, you must know both what is on the stack and where it is. How do you keep tally? You visit the nearest department-store kindergarten shop or contribute to 3M's dividend, that's what you do. More later.

Before we go on, a warning. When you work in the monitor and use the S or hardware stack, you won't find stack values starting at $0220, even though the S pointer says the S stack starts there; even though you PUSH values and PULL them accurately on the S stack. For example, run the program at left and then take a

```
ldd  #1
loop
  pshs  d
  incb
  cmpb  #6
until eq
leas 10,s
swi
```

good look at the S stack from $0200 to $0220. You'd surely expect to find 00 01, 00 02...00 05, in order, on the S stack. You won't. Yet if you add to the program at left to PULL the values and to store them, you indeed will PULL 00 01 through 00 05. Hmmmm. What happens? Revise the program at left. Begin it with: ldu #$6000, to start the user stack at $6000; change the "pshs d" to "pshu d"; and "leas 10,s" to "leau 10,u". Run the program and take a look at the U stack, from $5FFA through $6000. Well, well. The values we defined are indeed stored on the U stack as we expect, and right where they should be. Why can't we find them on the S stack? This is a most important question--for, suppose you run a test program using the S stack, and want to see if the values you expect are stored there. You won't find them. You can easily conclude that your program is incorrect when it isn't.

We bent our mind into pretzels on that problem before we asked Loch Rose what went on. He replied (chuckling, dammit) that any SWI (Software Interrupt), like all interrupts, stacks all register of the 6809 on the S Stack. Obviously! #@!! We ran our programs in the monitor; they always end with SWI. Those stacked registers overwrite whatever else was on the S stack. A breakpoint in the monitor is also an interrupt, so there is utterly no way to look at the S stack unless you store its values somewhere else. When you test a new program and must check the contents of the stack, run the tests on the U stack.

We return to how to keep tab on what's on the stack, wherever you start it and whichever stack you use--S or U. A notepad isn't of much use because the values continually change as you PUSH and PULL. You get tired of redrawing the whole stack and stack pointer with every change. We found a few simple answers, and track what's on the stack and where the Stack Pointer is with the diagram at left. There are four easy ways to keep your diagram current:

Relative Stack Position (bytes):		Stack Contents:
0		Parm 2 - Parm for subroutine
-2		JSR Return address from subr.
-4		Parm stacked by subroutine
-6	SP->	Second parm stacked by subr.

1. Buy a child's slate and a piece of of chalk. It's messy but easy to use.

2. Get a child's drawing platen, which is a clear plastic sheet over a waxy substrate. When you write on the plastic, it sticks to the wax and reveals what you wrote. Erase by pulling the plastic away from the substrate. Slit the plastic for the strips wanted.

3. Slit a small notepad with a razor. Erase by tearing off a strip.

4. Simplest of all, buy some 3M Scotch Post-It Pads, and cut 'em into strips which you can stick on or pull off without wrecking the writing underneath. Even better is Post-It Note tape; you can write on it; it comes in a dispenser, like Scotch Tape. Bless 3M.

Any value put on the stack stays there until it is overwritten--just as with any location in memory. If you stack a value, don't stack it again. If you keep tab of what's on the stack, you can get any value off it. You'd be surprised how often the same data are written to the stack, even by professionals who failed to keep track of what was already there!

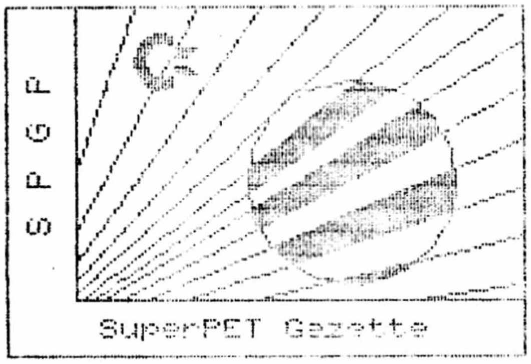
A GRAPHICS PROGRAM FOR SUPERPET AND THE 8023P PRINTER

High resolution graphics can't be had on SuperPET's screen without a new board or so. But that's no reason you can't create them on your printer, as we demonstrate elsewhere this issue. Delton B. Richardson of 4299 Old Bridge Lane, Norcross, GA 30092, has written a program, in part in 6502 machine language, which gives any ISPUGger the capability to design and print graphics to the limit of the resolution of the Commodore 8023P printer. No, you don't have to know maple syrup from machine language to use it.

He calls the program SPGP (for SuperPET Graphics Program). Why is it written in 6502 code? Well, Delton stores the 4K of machine language in high user memory; the rest of the program is written in BASIC 4.0, which also resides in user memory. This leaves all of the upper 64K in SuperPET available for the image. Each image contains 512 x 768 pixels (393,216) for a page of about 8.5x11 inches. Two character fonts are also stored in the upper 64 (you can change these). That's a lot of data, and almost fills 64K bytes.

The BASIC program is menu-driven, and allows you to design your own images and character fonts. You may combine both graphics and text and save images you have created to disk. Delton provides on disk the 6502 source code, which he wrote, assembled, and linked using WATCOM's 6502 Development System (which is very similar to the 6809 Development package in SuperPET, and runs on the 6809 side. You use the microEDITOR to write and edit).

Because Delton wants to distribute this disk as FREEWARE (more later), he has taken inordinate pains with his instructions and with the tutorial on disk; the BASIC program itself is user-friendly (in the best sense of that tired phrase). He conducts you through the program step-by-step until you thoroughly understand how to use it. Since the source code is available, you can modify the program to work on other printers; indeed, you can change it to become part of an application program. We print at left a small sample of some graphics Delton generated for us. Though the ribbon for Delton's printer is much healthier than most in ISPUG (we suspect all were new when Achilles chased Hector around Troy and haven't been inked since), the sample graphics at left are too light to reproduce well. We hope you can see and appreciate the detail.



As it stands, the program should be used only by those with 8023P printers. The material fills all but 86 blocks of an 8050 disk; it will not be made available in 4040 format because of the number of disks needed and because the program does not call for disk switches. Those who want the disk may obtain it from the Editor at PO Box 411, Hatteras, N.C. 27943 for $10 U.S. You are not only free to copy the disk and send it to friends, you are encouraged to do so.

If you like the program and its documentation, and feel the author should be rewarded for his work, Delton asks that you send a $20 contribution. You aren't obligated to do so (particularly if you don't like it!). Remember, however, that authors have little motive to issue good software with good instructions if they are not rewarded by a tangible token of your thanks (bullfeathers for money). If you expect more good software, you must flash a carrot. The Gazette will follow this FREEWARE experiment and report what happens.

ON VARIABLE FILES, PLAIN FILES AND HOW TO FOOL THE SYSTEM One of the three Waterloo disk filetypes is the VARiable file--most useful for saving machine-language code because you can write to and read from VAR files any value, including the CONTROLs (ASCII 0 thruough 31), as we noted last issue. The .mod files generated by the linker in SuperPET are formed as VAR files; we used them last issue to store PIC (Position Independent Code) since the CONTROLs may be loaded without CONTROLling SuperPET.

In APL, unlike the other languages, the default filetype is the VAR file. Workspaces are automatically saved as VAR,PRG files; APL-sequential and BARE-sequential files default to the format of VAR,SEQ, though you may change their format to TEXT,SEQ (and, indeed, APL creates BARE-sequential files in PRG format whenever you SAVE a workspace). The word "sequential" means that data are saved in the sequence filed, and not necessarily that the DOS format of SEQ is used.

You won't find any prefacing (v) for VARIABLE filetype on directory, any more than you'll find a prefacing (t) for the TEXT,SEQ files created by the micro-EDITOR, or an (f) preface for FIXED files in REL format. Indeed, as we'll see, the files do not know their own filetype (VAR, FIXED, TEXT).

Let us now examine the anatomy of VAR files, which depends upon where and how they are used. All of them contain data about the file in the first few bytes. A VAR,PRG file formed by the linker (a .mod file) will bear the following data in the first six bytes (the example below loads in bank 15, at address $9000, and is $97 bytes long):

Byte No:	1	2	3	4	5	6	7
	90	00	00	97	01	0F	Start of User Code
Means:	Load Address		Byte Length		Load in	Bank F	

VAR,PRG files so prefaced are meant to be loaded from main menu or in the monitor; at either place, the operating system loads them at the proper address and strips off the first six bytes after it has read them and knows what to do with the file. Obviously, you can load it anywhere by a change to bytes 1-2 or 5-6.

In contrast, a VAR,USR file you make to load into a language requires only two initial data bytes showing the length, in bytes, of the executable code in the file. The languages expect these first two bytes to show length, and will strip them off when the file is loaded. You can demonstrate this very easily by forming such a file and reading it both as a TEXT and as a VAR file. If you tell the language interpreters you have a TEXT file, you'll see the first two bytes in memory after you load the file; if you say it's a VAR file, you won't. In short, SuperPET and its interpreters do not know anything about filetype unless you specify it. You can pull some rather marvellous tricks once you realize this, as we'll later show.

In APL, a VAR file is far more complex in APL-sequential format. The difference between VAR,SEQ and TEXT,SEQ files in APL with BARE-sequential format is minor. A VAR file is created by default in APL with a program such as that at the left.

```
[ 1] 'DISK/1.VAR_BARE_SEQ' [] CREATE 4
[ 2] 'ABCDEF' []PUT4
[ 3] []UNTIE 4
```

We don't use a (v) to preface the device because VAR is a default. If we block-read the file in hex, we find:

```
00 61 62 63 64 65 66
```

If we change line 1 of the program to create a TEXT,SEQ file with: [1] '(T)DISK/1.TEXT_BARE_SEQ', the initial null is left out; we form a standard TEXT,SEQ file, which can be read by mED or in the other languages as such. So: if you want to convert a VAR,SEQ file to a TEXT, SEQ file without the hassle of running a program again, simply read the first byte of the VAR,SEQ file and throw it away.

The mBASIC manual says that mBASIC does not support VAR files; well, it does and doesn't. mBASIC can't form VAR files as such, but you can create them as TEXT files and then read them as VAR files--and mBASIC will dutifully strip out the first two bytes, though it will still consider any CR as End-of-Record and strip it out when the file is loaded.

All of this leads us to the stupidity (in kinder terms, ignorance) of the language interpreters in SuperPET regarding filetype. Let us demonstrate: put a short REL file, FIXED filetype, on a test disk. Suppose its name is "(f:128)test,rel". In the mED, copy it to another disk with: cop (f:128)test,rel to disk/1.newtest, seq. It copies. Then recopy it back to REL format with: cop disk/1 newtest,seq to (f:128)test,rel. Again, it copies. But--will the copied file now run as a REL file? Indeed it will! We compared the original file "test,rel" with the copied file "newtest,seq"--and with the final, copied "test,rel" on disk/1, using PIP. All three files, despite their differences in name, were identical.

When we thought about it, we whumped up an hypothesis: You can copy any file as a TEXT file (whatever its DOS format--PRG,REL,USR,SEQ) if you simply tell the dumb DOS and the dumber interpreters that the file you want copied is a TEXT file. The DOS copies any file identified as a TEXT file as-is; the prefacing bytes in VAR files and the side sectors in REL files read and transfer just as they stand. Example: we copied a file "(f:74)test,rel" to another disk as file (t)disk9/1.test,rel in the mED; checked it with PIP (files identical!), then recopied it to fixed filetype, and it ran perfectly. In short, our hypothesis is confirmed: you can copy any file as a TEXT file if you use the right DOS suffix:

(f:80)program,PRG may be copied as (t)program,PRG
(f:128)test,REL may be copied as (t)test,REL
(v)trial.mod,PRG may be copied as (t)trial.mod,PRG
(v)pic,USR may be copied as (t)pic,USR, etc.

What may we then conclude?

1. The filetype identification demanded by the interpreters (as FIXED, VARIABLE, or TEXT) merely tells the interpreters how to read or how to format the bytes in the file. We include the mED in the term "interpreters."

2. The files themselves are merely a series of binary bytes written in an arbitary format to arbitrary rules set up by the DOS and the interpreters. You may evade the rules when it is convenient. If you cannot remember the record length of a REL file, copy it as a SEQ file, read it in mED and find the record length. Or, should you want to modify a VAR file, read it as a TEXT file, amend it, refile it as a TEXT file, and read it as a VAR file (see II, p. 44).

3. Next, you may use assembly language to copy any file, in any filetype, byte for byte, if the routine you write considers that every file copied is a TEXT file. You need worry about only one thing: that the file has the right name after being copied.

4. In sum, the filetypes are set up so that the interpreters and the mED know 1) how to format a file, and 2) how to read it. You may quite easily establish your own USR format for any file, writing and reading it as is convenient. After all, that is exactly what the DOS and the interpreters do!

SEARCH AND REPLACE IN THE MICROEDITOR
Definition and Simplification

The explanation in the Systems Overiew manual on this subject is terse and hard to follow. We summarized the process in Reference Sheet 3 of the Starter-Pak manual, but find even that summary far from complete. This article attempts to define command characters, metacharacters, and the search and replace functions simply, clearly, and in depth.

Before we proceed, a warning: it often is far simpler and safer to manually correct a few phrases than to parse and write a complex find/change command. If you require a massive number of changes, you may well enter and execute an bad one, which will destroy your program. Test on a few lines first!

SEARCH COMMANDS: These are limited to the slashbar used alone or prefixed by + or -, as shown below. The first slashbar means "search for"; the second is a terminator for the search string.

/ word /	Find the first occurrence of "word", prefixed and suffixed by at least one space; start search at first line of file in memory.
+/	Assumes a search command has been uttered, as above. Finds the next occurrence of "word". Repeats a search for the last search string uttered as often as the command is entered. Searches from current line (of screen cursor) downward.
/phrase/	Find a defined phrase, with or without terminal spaces or within a larger string. Finds "fix" within "suffix" or "demo" within the words "demon" or "demonstration"; also finds " demo ".
-/	Repeats the last search command entered, searching from the current line to start of text.
-/ word	A variant of / word/. The absent terminal slashbar creates a null after "word" in the search string.
/ /	Finds two (or more) spaces, but no blank lines or terminal blanks, which are NULLS, not spaces.

SEARCH LIMITS: Though you may with other mED commands define a line range for the command, as in "12,20 p disk", you may not define a specific line range for search: "1,20/e", an attempt to find an "e" in lines 1 through 20, fails. Only + and - specifiers are allowed. The false command "5,/ " will execute, but finds nothing, scrolling the screen from line 5 to end of text. We don't know why line range specifiers are absent from the search command, since we do have them with the search/replace command, and we could speed up searches if we were allowed to place line range specifiers. We wish we had 'em. [Ed. In the latest version of Joe Bostic's BEDIT, they're emplaced!)

Nulls and Spaces: SEARCH itself is more complex than might appear. Let us try to search the last two lines of the previous paragraph for /place/. We'll find it twice, once as a terminal phrase in "emplaced". If we want only " place ", we might amend the command to add two spaces: / place /. Unfortunately, that will not find "place " at the start of any line of the last paragraph--the start of a line isn't a space. We need: /place / to find it--but that form of search won't find " place" at end-line; the terminal blank there is a null, not a space. At end-line, we must search with / place/. To our sorrow, that command also finds words such as placebo and placenta.

In short, the SEARCH command lacks a way to find a word bounded by a null or by a space to catch search strings at start and end of line. We need a metacharacter to designate a null or space; with it, search/replace would be universal and much more powerful. Suppose Waterloo had used a tilde, thus: /%~place%~/, where %~ means: "a space or null." We hope those who get their kicks writing new version of the mED take this suggestion to heart! (Ed. Later comment: We now have this metacharacter in BEDIT; it represents not only the "null or space" we asked for, but also any non-alphabetic characters. You may now find and replace every occurrence of a specific word or phrase--at end of line, start of line, or surrounded by non-alpha characters, as in: (placeit) and placeit=2. Our thanks to Joe Bostic for this ingenious extension. Search and replace now is universal!)

Part of the Search/Replace problem may be our own fault. If we write code such as, "limit=limit+2", we cannot search/replace the variable name "limit" without also finding and changing every "limited", "limits", and "delimit" in the program. If code is written with conventional spaces (limit = limit + 2), then search and replace is simple and safe. (Ed. See comment above; the %~ meta in BEDIT now solves this problem as well.)

COMMAND CHARACTERS and METACHARACTERS: Some of the Command Characters (the ., $, and *) serve both as Command Characters and as Metacharacters. They play entirely different and distinct roles in these two functions:

Command Characters and Meaning:		Metacharacter Meaning:	
.	Line where screen cursor rests.	%.	Represents any character.
$	The last line of a file.	%$	Represents the end of a line.
*	Global command.	%*	Represents zero or more repetitions of the character preceding.

The characters above often intermix, in the two roles, in the same command. With these differences in symbols in hand, we review the metacharacters, which are formed with an initial % coupled to one or two added characters. Metacharacters are easier to understand if parsed in doublets or triplets. See below:

Metacharacter and Meaning:		Comments:
%	Represents any character	Do not use alone.

%%	Represents a "%"	Will find and replace a single %.
%/	Represents a slashbar "/"	A special way to find / is required, since it is the terminator for SEARCH and REPLACE.
%^	Represents start of line	Since the carat or up arrow defines the start of a line, you need a special way to find %^, as shown at left.
%%^	Represents a "%^"	
%.	Represents any character	Useless unless used in combination. Since . has a special meaning, you must have a way to find . preceded by %, as at left.
%%.	Represents a "%."	
%$	Represents end of a line	Search strings must precede this metachar.
%*	A wild card call; represents ZERO or more repetitions of the character which precedes the %*; useless unless used in combination.	
%&	Represents, in a change string, the text found by the SEARCH command. It is useful to change complex phrases, as in: *c*/ zpat24s / mod_%&/, which will change "zpat24s.1" to read "mod_zpat24s.1". It avoids copy errors.	

THE CHANGE COMMAND: There are four forms of the change command: *c* changes all occurrences on all lines. *c changes all first occurrences on all lines; c* changes all occurrences in the current line; c changes only the first occurrence in the current line.

Only c* and c may be used with line range commands (both *c and *c* are global commands). Thus 1,20 c*/... and .,+20 c/... and similar commands are allowed.

Search and Replace Examples We illustrate some useful commands:

Command:	Comments:
*/:bd/d	Delete all LINES holding ":bd". We are indebted to Terry Peterson for this. It globally finds and deletes any line containing the phrase ":bd". It combines both the search and delete commands. Superbly powerful.
/variable%%	Finds the integer "variable%"; most useful in microBASIC.
c/ x%% / flag%% /	Globally changes the integer variable x% to flag%.
/10-%.%*-83	We search a list of dates for entries in October, 1983. The first %. searches for zero or more repetitions of any character; the %* for zero or more repetitions of the preceding "any character" designator. Finds entry for any day in the month or year specified, as in: 10-12-83 or 10-1-83. God help those who write dates with slashbars, as in 10/21/83! Parse that one!
*c/%^"/	Finds any quotation mark at the start of a line and replaces it with a NULL, pulling all text left one space. Useful for removing the quotation marks from WordPro files converted to ASCII for use in 6809. Very slow on long files. Be patient.

A second slashbar, as in *c/%^"//, will work but isn't needed.

*c/ / Globally deletes two spaces, if present, at the start of every line; the splendid "indent remover." A final // is not needed.

.,+10 c*/ /   / From current line forward ten lines, change all single spaces to three spaces. Superb for spreading monitor dumps out for comment or for widening titles and captions.

Automatic Copying of Files Using Search/Replace

Assume we have on screen a disk file of a directory, in which a typical entry is as follows: 12   "patch:e"   SEQ .

c/%.%"/C1:*=0:/ At start of all lines, we replace everything through the first quotation mark. The command at left changes all entries on the screen so that the file title follows the Copy command:

```
C1:*=0:patch:e"    SEQ
```

c/"%.%/ Let us then utter the command at left; we delete everything from the quotation mark (above) through end of line, leaving this typical entry on screen:

```
C1:*=0:patch:e
```

If we then put the screen cursor on the line of each file we want copied, and say: . p disk at command cursor, the DOS will copy each file without further typing. The batch file capability emerging in new programs written by our members takes advantage of this approach by putting the search/replace commands into a disk file which can be executed automatically any time you EXEC it. We write these commands once; they are forever available when we must process a number of files selected from a large directory. We thank Terry Peterson for the last two examples.

If you take the process one step further, you may create a BATCH file which will automatically EXECUTE the COPYing from one disk to another.

THE APL EXPRESS by REG BECK

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

In my previous two columns, I've devoted space to APL system commands and functions, and to manipulation of files. Further odds and ends are included here.

THE FOLLOWING EXAMPLE SHOWS THE DIFFERENCE BETWEEN)SYSTEM COMMANDS AND ⎕ SYSTEM FUNCTIONS: LIB REFERS TO LIBRARY, THE APL VERSION OF DIRECTORY.

```
)LIB            ⍝DISPLAYS DIRECTORY FROM DISK 0 IN DIRECT MODE ONLY.
)LIB DISK/1     ⍝DISPLAYS DIRECTORY FROM DISK 1 IN DIRECT MODE ONLY.
⎕LIB 'DISK'     ⍝DISPLAYS DIRECTORY FROM DISK 0 IN DIRECT OR FUNCTION MODE.
⎕LIB 'DISK/1'   ⍝DISPLAYS DIRECTORY FROM DISK 1 IN DIRECT OR FUNCTION MODE.
```

IN EACH OF THE ABOVE CASES DEVICE NUMBER 8 IS ACCESSED. TO ACCESS ANY OTHER DEVICE, USE DISK9/1 FOR INSTANCE FOR DEVICE NUMBER 9. SYSTEM COMMANDS NEVER REQUIRE QUOTES. SYSTEM FUNCTIONS ALWAYS DO. A FURTHER EXAMPLE OF THIS IS:

```
)LOAD SORT      ⍝SORT IS A SAVED WORKSPACE ON DISK AND BECOMES THE ACTIVE
                ⍝WORKSPACE, REPLACING ANY PREVIOUSLY ACTIVE WORKSPACE.
⎕LOAD 'SORT'    ⍝IS IDENTICAL TO )LOAD SORT AND MAY BE USED IN DIRECT MODE
                ⍝OR AS AN APL STATEMENT IN A USER DEFINED FUNCTION.
```

The usual method is to use the) system commands (if there are appropraite ones) in direct mode (from the keyboard), and to use the quad system functions mainly in defined functions, or to manipulate files.

```
TEXT MAY BE DUMPED TO A PRINTER USING THE FOLLOWING PRINTER NAMES AS FILE
NAMES.  EXAMPLES FOLLOW.
'PRINTER'       ⍝CBM PRINTER, ASCII IS CONVERTED TO PET ASCII.
'IEEE4'         ⍝ASCII PRINTER ON THE IEEE488 BUS USING DEVICE NR. 4.
'SERIAL'        ⍝ASCII PRINTER ON THE RS232 PORT.

'IEEE4' ⎕CREATE 1        ⍝OPENS A FILE TO THE IEEE4 PRINTER AND TIES IT.
(⎕LIB 'DISK') ⎕PUT 1     ⍝PRINTS THE DIRECTORY FROM DRIVE 0 ON THE PRINTER.
        ⎕UNTIE 1         ⍝UNTIES THE PRINTER FILE.

EX←'NOW IS THE TIME FOR ALL GOOD MEN, ETC.'  ⍝ASSIGN SOME TEXT TO EX
'SERIAL' ⎕CREATE 5       ⍝OPENS A FILE TO A SERIAL RS232 PRINTER.
      EX ⎕PUT 5          ⍝PRINTS EX.
         ⎕UNTIE 5        ⍝UNTIES THE SERIAL PRINTER.

    'IEEE4' ⎕CREATE 1    ⍝OPENS A FILE TO IEEE PRINTER.
 '(T)STUFF' ⎕TIE 2       ⍝OPENS A TEXT FILE STUFF ON DISK 0.
(⎕GET 2 79) ⎕PUT 1       ⍝GETS 79 BYTES FROM STUFF AND PRINTS IT.
(⎕GET 2 79) ⎕PUT 1       ⍝GETS NEXT 79 BYTES FROM STUFF AND PRINTS IT.
(⎕GET 2 79) ⎕PUT 1       ⍝GETS NEXT 79 BYTES FROM STUFF AND PRINTS IT.
            ⎕UNTIE 2 1   ⍝UNTIES BOTH PRINTER AND DISK FILE.
```

The (T) in the filename indicates a text file. If you wish to communicate with the outside world from APL, be sure to prefix all filenames involved with (T).

* * *

Direct function definition is a method of writing APL functions which appears more and more frequently in texts and articles. When you use direct definition, you enter information about the function to be defined; it is transformed into the required APL function by a compiler, a set of functions for this purpose. The compiler I use was written by Ted Edwards of Capilano College. It allows you to define recursive and conditional functions.

Loading Ted's compiler into the active workspace displays the following on the screen:

```
COMPILER FOR DIRECT DEFINTION BY TED EDWARDS
PERMITS RECURSIVE USE AS IN


      DEFINE
FAC : ω × FAC ω-1: ω=1 : 1
FAC
      FAC 5
120

DEFINITION READS
```

FAC IS DEFINED AS ω × *FAC* ω-1, *UNLESS* ω=1 *IN WHICH CASE*
FAC 1 *IS DEFINED TO BE* 1.

FAC is the factorial function, in this case recursively defined as an example of this type of definition. FAC 5 returns the product of all positive integers up to 5. A simpler non-recursive example is given below (FACN). As defined, FAC is a monadic function. Omega, used in the definition, will be replaced by the right argument of the function. In the definition of a dyadic function, omega is used for the right argument and alpha for the left. ROUND is an example of a dyadic function in direct definition. To invoke the compiler, the word DEFINE is entered and the cursor moves down one line and over to the leftmost screen column. The function name is typed in, followed by a colon and the definition. When this has been entered and returned the compiler does its job; if it succeeds, it prints the name of the compiled function; otherwise an error message appears.

```
FACN: ×/⍳ω                     ⍝NON-RECURSIVE FACTORIAL.
      FACN 4
24
MEAN: +/ω÷⍴ω                   ⍝FINDS MEAN OF A VECTOR.
      MEAN 1 2 3 4 5
3
ROUND:α×⌊.5+ω÷α                ⍝ROUNDS OFF TO ACCURACY INDICATED BY α.
      .01 ROUND 52.34698
52.35
```

I include a listing of the compiler functions except for DESCRIBE, which produces the description in the APL text above. I print these listings reluctantly, as they take up a whole page. Some of you, however, may have some willing slaves (i.e., students) who could type in the listings; otherwise, you'll have to acquire a compiler on disk. The program is in the public domain.

```
      ∇DEFINE[⎕]∇
[  0]    R9 ← DEFINE ;I99;I9;NM9;SPL9;NCL9;CR9;DFN9;V9
[  1]   ⍝820224 COMPILES DIRECT DEFINITIONS . BY EMEDWARDS
[  2]   ⎕IO←1
[  3]   I9←⍞
[  4]   →0×⍳0=⍴I9←((∨\I99)∧⌽∨\⌽I99←' '≠I9)/I9
[  5]   NM9←(⌽∨\⌽' '≠NM9)/NM9←(SPL9←¯1+I9⍳':')↑I9
[  6]   →ERROR9 ⌈⍳ (⍱/NCL9←0 3=⎕NC NM9)∨'9'∊NM9
[  7]   →ERROR9 ⌈⍳'DEFINE'∧.=6↑NM9
[  8]   →((' '∧.=(1+SPL9)↓I9)∧NCL9)/ERROR9,SHOW9
[  9]   →ERROR9 ⌈⍳0÷×/⍴R9←COMPILE9 I9
[ 10]   →ERROR9 ×0=0\0⍴ R9←⎕FX R9
[ 11]   SHOW9: →ERROR9 ⌈⍳ 3 5 ∧.≠1↑⍴CR9←⎕CR NM9
[ 12]   → ERROR9 ⌈⍳ (⍝≠1↑DFN9)∨≥/(DFN9←CR9[2;])⍳':'''
[ 13]   ⍎(1=1↑⎕FX CR9)∧V9←':'∊I9)/'→ERROR9,0⍴⎕←''NO! FUNCTION SUSPENDED'''
[ 14]   →1 ⌈⍳ V9 ∧' '∨.≠I9
[ 15]   →0,0⍴⍎(~V9)/'R9←1↓DFN9'
[ 16]   ERROR9:'DEFINITION ERROR',0⍴⎕EX 'R9'
      ∇COMPILE9[⎕]∇
[  0]    R ← COMPILE9 X ;L;Z;I;J;IN;LOC;ARG;HED
[  1]   ⍝CREATES CANONICAL FORM FOR RECURSIVE DIRECT DEFINITION
[  2]   ⍝E.M.EDWARDS
[  3]   R←0 0 ⍴Z←I\(I←L∧=\L←''''≠X)/X
[  4]    ARG←2 1⌈.×∨/L←'αω'∘.=Z←(I←∧\Z≠'⍝')/Z
[  5]    LOC← LV9 Z
[  6]    Z[(∨⌿L)/⍳⍴Z←I/X]←'LR'[(0≠J)/J←1 2+.×L]
[  7]    Z←(Z,(⍴IN)⍴'9')[⍋I,IN←(∨⌿L)/I←⍳⍴Z]
[  8]    →0 ×⍳∧/3 5≠1↑⍴Z←':' B9K Z,':⍝'
[  9]    HED← 'Z9← ',((2=ARG)/'L9 '),((⌽∨\⌽' '≠I)/I←Z[1;]),((0≠ARG)/' R9'),L
[ 10]    ⍎(IN<I←⌈/(⍴HED),(1+⍴X),9+IN←1↓⍴Z)/'Z←Z,((1↑⍴Z),I-IN)⍴'' '''
[ 11]    Z[1;⍳⍴HED]←HED,0⍴Z[''⍴⍴Z;1+⍳⍴X]←X
[ 12]    →COL1 ⌈⍳ 3=1↑⍴Z
[ 13]    Z[;]← 0 0 ¯4 ¯9 ¯9 ⌽Z[1 5 3 2 4;]
[ 14]    Z[4 5;⍳8]←2 8⍴'→0,0⍴Z9←',0⍴Z[3;⍳4]←'→4⌈⍳'
[ 15]    →0,0⍴R←Z
[ 16]    COL1: Z[;]←0 0 ¯4 ⌽Z[1 3 2;]
[ 17]    Z[3;⍳4]←'Z9← '
[ 18]    R←Z
      ∇LV9[⎕]∇
[  0]     Z ← LV9 S ;ASS;VAR;A;GET;E;L
[  1]    ⍝PUT LOCAL VARIABLES IN HEADER FOR DDEF
[  2]    Z←''
[  3]    →0×⍳1=+/ASS←S∊S[S⍳':']←'←'
[  4]    VAR←S∊'ABCDEFGHIJKLMNOPQRSTUVWXYZ_'
[  5]    GET←VAR∧A∊(VAR∧1⌽ASS)/A←+\VAR<1↓VAR,0
[  6]    Z←(4≠⎕NC Z)⌿Z←'←' B9K (GET∨ASS)/S
[  7]    Z←(0,1↓(A⍳A)=⍳⍴A←(Z∧.=⍉Z)⌈.×⍳1↑⍴Z)⌿Z
[  8]    Z←(' '≠Z)/Z←,';',Z
      ∇B9K[⎕]∇
[  0]     Z ← C B9K X ;L;N;E;⎕IO
[  1]    ⍝ BREAK VECTOR X INTO MATRIX ON DELIMITERS C IGNORING
[  2]    ⍝ C'S IN QUOTES.  E.M.EDWARDS
[  3]    L←(X∊C)∧L∧=\L←''''≠X
[  4]    N←(N,⍴L)-0,1+N←L/⍳(⎕IO←0)+⍴L
[  5]    Z←(⍴E)⍴(,E←(N-1)∘.≥⍳⌈/N)\(~L)/X
```

Printed below are further examples using Ted's compiler:

```
MAT:(ω↑V),[1]MAT ω:0=⍴V←⍞:(0,ω)⍴' '
FMAT:(ω↑((ω-⍴V)⍴' '),V),[1]FMAT ω:0=⍴V←⍞:(0,ω)⍴' '
THE FIRST FUNCTION, MAT, FROM TAMA TRABERMAN'S ARTICLE DISCUSSED NEXT,
PERMITS DIRECT ENTRY OF A VERTICAL LIST.  FMAT, WHICH I OBTAINED BY
MODIFYING MAT, RIGHT-JUSTIFIES A LIST AS IT IS ENTERED.  EXAMPLES:
CONTINENTS←MAT 15        ⍝USING MAT, THE CONTINENTS ARE ENTERED DIRECTLY
EUROPE                   ⍝MAKING ENTRY EASIER FOR STUDENTS.  JUST
ASIA                     ⍝TYPE IN EACH ENTRY AND HIT RETURN.
NORTH AMERICA
SOUTH AMERICA
AFRICA
ANTARCTICA
AUSTRALIA
POPULATIONS←FMAT 12      ⍝FMAT SETS UP A CHARACTER MATRIX OF
490171000                ⍝RIGHT-JUSTIFIED POPULATION NUMBERS AVOIDING
2647970000               ⍝POWER OF TEN NOTATION WHICH WOULD BE
249722000                ⍝UNSUITABLE FOR USE WITH ELEMENTARY STUDENTS.
383188000                ⍝THIS CANNOT BE AVOIDED WITH LARGE NUMBERS IF THEY
49696000                 ⍝ARE ENTERED AS A NUMERICAL VECTOR.
```

```
        ldd #string1
        pshs d
        ldd number
        pshs d
        ldd #string
        jsr printf_
        leas 8,s

; same example using PIC methods

        leax string2,pc
        pshs x
        ldd  number,pc
        pshs d
        leax string1,pc
        pshs x
        ldd  number,pc
        pshs d
        leax string,pc
        tfr x,d
        jsr printf_
        leas 8,s
        swi
string  fcc "The number %d %s is %h when %s to hex.%n"
        fcb 0
number  fdb 55
string1 fcc "in decimal "
        fcb 0
string2 fcc " converted "
        fcb 0
        end
```

3 P3 is the address of 'in decimal.'
4 P4 is the value of number.
5 P5 is the address of 'converted.'

The items must be placed on the stack in the order: P5, P4, P3, P2; then P1 must be placed in D register before we call the routine PRINTF_.

Because P1, P3, and P5 are addresses we use LEAX to determine them. Note that only the immediate mode of addressing, used otherwise to load an address, must be changed to employ LEAX. LDD #55 in this example would substitute nicely for the LDD number,pc instruction. The TFR operation is required only because P1 must be placed in the D register.

It is not at all difficult to make your programs use the Waterloo library in PIC code. Just remember how to use the LEAX (or, for that matter, LEAY) instruction, and let the assembler worry about the offset.

The difference with PIC is obvious: The code required to write an application in PIC must be longer than similar code using absolute addresses. Yet a routine which has been so encoded will operate properly no matter where it is located, while a routine using absolute addresses must be drastically altered every time it is relocated.

You may assemble a PIC routine once and forget it. If you later want to move the code you may use the MOVE instructions from either SPMON or XMON-6809 and the code will still click.

So, here we have shown just how easy it is to move from absolute coding of rou-routines to writing PIC. With a little practice it will become quite natural. For those wanting a good example: disassemble some code of SPMON and XMON-6809. Note that Terry uses PIC for SPMON, while XMON employs absolute addressing. To get HI and LO versions of SPMON, all Terry had to do was move the code to load at a new address. Several bugs were corrected in XMON6809 at the last minute, as patches, so a correct .asm file of the final version doesn't exist. Thus I can't relocate XMON, even by changing the ORG in a .cmd file. Now, you decide which method of coding is more effective... The case in favor of PIC is a strong one.

PC or PCR? IS THERE A DIFFERENCE? From reading Lance Leventhal, "6809 Assembly Language Programming," you will conclude that there is a distinct difference between specifying addresses ",PC" or ",PCR". You can infer from pages 3-23 thru 3-25 that ",PC" addressing is limited to absolute values (as in: LDD $03,PC) and that the programmer must do the math to determine the offset to the address he wants. At least that's the way we read what the man says--especially his comment on the programmer doing the arithmetic on p. 3-25, "We would like to have the assembler handle this for us, since the procedure is simple to explain but difficult to perform correctly." Leventhal then comments that the "standard 6809 assembler will calculate the program counter offset for you if you designate the address as 'program counter relative,' or PCR."

We therefore expected the Waterloo Assembler to react differently when we used PC and PCR. It didn't. We assembled some test code PC and PCR, and then compared the object code generated in the two modes. Both forms were accepted; the code was identical for the codings at left.

```
leax label,pc
leax label,pcr

ldd  $03,pc
ldd  $03,pcr

ldd  [label,pc]
ldd  [label,pcr]
```

It 'twould appear that Waterloo treated PC and PCR as identical modes, and that the SuperPET Assembler will generate identical code whichever mode is used.

This may not be true of code written for OS-9, or for other 6809 assemblers, where there may be a distinct difference as to what the assembler accepts in these modes. After discussing this with John Toebes, we recommend you use PC mode for absolute addressing (LDD $03,PC) and PCR for relative addressing (LDD LABEL,PCR)--exactly as Leventhal defines them. If you disagree, tell us why! We seek enlightenment.

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	No. 3: $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. SuperPET is a trademark of Commodore Business Machines, Inc.; WordPro, that of Professional Software, Inc. Contents of this issue copyrighted by ISPUG, 1985, except as otherwise shown; excerpts may be reprinted for review or information if the source is quoted. TPUG and members of ISPUG may copy any material. Send appropriate postpaid reply envelopes with inquiries and submissions. Canadians: enclose Canadian dimes or quarters for postage. The <u>Gazettte</u> comes with membership in ISPUG.

ASSOCIATE EDITORS

Terry Peterson, 8628 Edgehill Court, El Cerrito, California 94530
Gary L. Ratliff, Sr., 215 Pemberton Drive, Pearl, Mississippi 39208
Stanley Brockman, 11715 West 33rd Place, Wheat Ridge, Colorado 80033
Loch H. Rose, 102 Fresh Pond Parkway, Cambridge, Massachusetts 02138
Reginald Beck, Box 16, Glen Drive, Fox Mountain, RR#2, B.C., Canada V2G 2P2
John D. Frost, 7722 Fauntleroy Way, S.W., Seattle, Washington 98136

Table of Contents, Issue 3, Volume II

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

ELIZABETH CITY N.C. MAR 11 '85
U.S. POSTAGE 56
PB METER P.B.307549

First-Class Mail
in U.S. and Canada