SUPERPET GAZETTE

We apologize to readers for a blooper last issue in the assembly-language screen dump we published. While it dumps very well, it also crashes all languages if called with a SYS call, or if you do not overtype the registers in a monitor call. Sorry; it came in one day before we had to print and we didn't have time for a good test. No excuses; we should not have published without a good check, and normally don't. Here's how to make the dump run from the monitor only (don't use a SYS call from language): overtype all registers with 0 except for S and CC as shown, left, and, then hit RETURN. Do this after every dump, when the registers are automatically dumped to the screen. Second, do not give the command: clear <RETURN> unless you reset top of user memory to 7fff. If you do these things, the program works very well. By way of compensation, we offer you this issue a pluperfect, kandy-koated assembly language dump which runs in all SuperPET languages and facilities, from Development through APL, from the microEDITOR thru the MONITOR; which resets top of user memory in program, and which we've used for a solid month, every day, without a glitch. The dump lives up high at $7f00. You call it when you drop a line of forty backslashes '\', starting at left margin. Whereupon the program dumps to the line of backslashes, erases them, and stops.

```
  S      CC
 0220    c0
```

We can thank Jeff Larson and Gary Ratliff for it. Jeff wrote a new dump which is gentle with printers (it line spaces all blank lines instead of slamming from stop to stop), and does a carriage return at the end of the characters on any line. Gary, meanwhile, found a way to use the interrupt process in SPET to sense the dump command and to avoid the troubles we described above. When we got the two new versions, we asked Gary to perform a marriage (which he did). The result is PIRQ (Print on Interrupt). We've run it in all languages, and find no bugs. Yes, you can pick another key to start the dump, if you don't want to use the backslash.

You load it in the mornin' from the language disk with a simple: pirq <RETURN> in about two seconds; it returns you to menu; whereupon you load whatever language or facility you want. You may change languages or facilities and PIRQ remains in memory until you shut down or switch to 6502. See Gary Ratliff's article, in this issue.

****

A GAZETTE SPECIAL! THE FIRST PUBLISHED MEMORY MAP OF SUPERPET!

See Article, page 81, this issue.

****

THE EDUCATIONAL LIST Last issue, we asked if teachers were interested in a list of educators who use SPET, mutual aid being the idea. Jim Swift (RR#3 Site E, Nanaimo, B.C. Canada V9R 5K3) tentatively plans to handle a newsletter/educational list for educators using SuperPET. Unless we hear from teachers on our mailing list to the contrary, we will send your name and address to Jim as a start for his mailing list. If the address label on this issue of the Gazette does not show you at a college, high school, or university, we have no way of knowing you teach--so please send your name and address to Mr. Swift directly. To get help or give it: communicate!

****

JIM SWIFT'S GEM Last issue, you read that a 'g ieee8-15.NO:newname,id' issued from the microEDITOR (mED) worked as a DOS command, and that all DOS commands could be issued in this form from the mED wherever it is used. One problem, though: how do you handle filenames containing blank spaces? See the example at left. Any other arrange-

```
g 'ieee8-15.CO:space file=1:space file'
```

ment of quotes we've tested will fail. This works. The mED needs quotes around filenames only when blank spaces appear in that filename, as in: g 'blank file'.

*   *   *

NEW PUBLICATION SCHEDULE: REVISED MEMBERSHIP DUES Starting with this issue, we're increasing the size of a regular issue to about 20 pages, and publishing once each two months, instead of once each six weeks, because the production and distribution job is a horror. Instead of giving members about 80 pages a year, we'll give you about 120. Dues must go up to $15.00 U.S. for the U.S. and Canada to cover postage and printing [80:120 as 10:15]. Those who gave us multi-year memberships will continue to receive the Gazette until memberships expire. Overseas dues (Europe, South America) will change to $25.00 U.S. from $20.00, since the new arrangement economizes on foreign postage. We'll have to cut pages or membership periods if the Postal Service raises postage during the next twelve months, or if our printing goes up.

*   *   *

NOTES FROM THE SIEVE Last issue, we promised some notes on what runs fast and what is slow from extensive tests of mBASIC in the Sieve of Eratosthenes. The results are comparative, not absolute. Use the comparisons with judgment; the Sieve has only 26 lines; the comparison of 'goto' with 'if...else...endif' shouldn't hold for long programs.

SLOWER:	FASTER:	
if...else...endif	goto	
goto	if...endif	
20 prime% = i%+i% 30 k% = i%+prime%	Combine equalities on one program line.	Significant difference.
while...until <cond>	while...endloop	Significant difference.
if flag% = 1	if flag%	Substantial difference.
for i% = 1 to n (or 0 to n)	for i% = p% to q%	Measurable difference in 8190 iterations.

loop...endloop
while...until
if...endif
	for i% = p% to q%	Most significantly and substantially faster.

Lines numbered 1,2...26 ran not a whit faster than lines numbered in increments of 5 or 10 [10,20...260]. In 8190 iterations, removing the white spaces between program items [count% = 0 vs. count%=0] decreased run time only 2.6%--which is hardly worth the decrease in readability. As a last test, we took out ALL indentations for structure. For a grotesque loss in readability, we cut run time only 1.8%. So let no contributor send in programs written BASIC-style, with everything crammed together, on the excuse that 'it's faster.' Significantly, it is not significant. Congratulations to Waterloo for making such measures unecessary. Now, if we just had compilers....

*   *   *

****

INTEGRATING TEXT AND PROGRAMS IN THE MICROEDITOR The assembly-language screen dump printed last issue was significant, for it'll add the MONITOR as the last, reluctant facility or language in SuperPET whose programs and output can be integrated in the microEDITOR. All we need to finish the job is a rewrite of the assembly-language dump so it saves to disk (primarily to store monitor dumps).

The high-level languages which run in 6809 mode are no problem. Any sequential file generated by SuperPET, in any 6809 language, can easily be brought from disk into the mED. The problems: how to pull WordPro, Wordcraft, and BASIC 4.0 files into the mED, how to print APL files integrated with ASCII files, and how add to, abridge, page, file to disk, and put to printer in one pass the files we thus create, of whatever length. It's a tall order, but a large part of it is done. We'll cover part of it this issue, and more next, being short on space.

What follows is the result of collaboration between Gary Ratliff and the editor; just who did what is not at all certain; as with Topsy, the methods 'growed,' from a suggestion by Gary last December, some great assembly-language work from him, and five months of interesting work.

* * *

WARNING! Strongly suggest you not file WordPro, Wordcraft, or BASIC 4.0 files with any capital letters in the filename. When read in 6809 mode, capitalized filenames appear in reverse field, and files can neither be retrieved nor manipulated by any possible disk command in 6809. To use the methods we describe, avoid caps in 6502 filenames like plague.

* * *

CONVERTING BASIC 4.0 PROGRAMS FOR THE mED

The microEDITOR reads only sequential (character) files. BASIC 4.0 programs 'dsaved' are program files, and thus not readable. You can convert them to sequential files if you 'dload' them and save them to disk while in BASIC 4.0 with the immediate mode commands on the left; they save a sequential file to drive 0, with a new filename. Wait until the cursor reappears to enter the second line.

```
open 5,8,4,'0:filename,s,w':cmd 5:list
print#5 : close 5
```

When a file is so converted to a sequential file, you cannot load it in BASIC with 'dload', but if you load the mED alone in 6809 (outside any language), and say: g 'FILENAME' <RETURN>, the file will load into the mED. Be sure to use CAPITAL letters for the filename, or mED will refuse your command.

If you have used cursor commands within quotes in the BASIC program, they will not print properly in mED. You have two choices: type in the COMPUTE! conventions for hard copy cursor commands before conversion, or use Gary Ratliff's machine language program to convert them to the conventions (in English) automatically. Gary's LIST-9 utility does it almost instantly, either to screen or disk. His program is found on the SPUG disk announced last issue. You need enter nothing in the monitor.

Here's how to use LIST-9: (1) load LIST-9 from drive 0 with: load '0:list-9',8 <RETURN>; (2) reset BASIC pointers with: new <RETURN>. Then 'dload' the program you want to convert. When it's loaded, issue the command below, left (program is set to file to drive 0).

```
open 5,8,4,'0:filename,s,w':poke 78,n:cmd 5:sys 40539
print#5 : close 5
```

The value of 'n', after the poke, is the number of characters you want on a line--the maximum being 80, or a full screen. After the command is entered, your program goes to disk with cursor commands in English. If you want to see the program on the screen, as converted, simply say: sys 40539 <RETURN> and you will have it, on screen, instantly. LIST-9 is a jewel, particularly when you must convert BASIC utilities to mBASIC utilities. BASIC programs thus converted may be loaded in the mED in mBASIC and rewritten.

[Next installment: Converting BASIC 4.0 files for WordPro; WordCraft and Wordpro files for use in 6809; handling giant text files (virtual and real).]

PRINT DIRECTORY - A Demonstration of Some Error-Processing Techniques and of Using Program-Controlled Immediate Mode in microBASIC

(c)

by John A. Spencer, Chemistry Department, Southern Illinois University, Edwardsville, Illinois 62026

[This, the second of Dr. Spencer's articles for the Gazette, should be mastered by every user of Waterloo microBASIC, for it not only demonstrates the basic error-processing techniques in SuperPET, but it also shows how to use what the old PET hands call the 'dynamic keyboard', in which the computer gives itself instructions. Last, it is a mighty handy utility, producing multi-column printouts of large directories, instead of the usual Commodore string of spaghetti.]

PRINT DIRECTORY accomplishes the difficult: printing a directory in mBASIC 1.0. It primarily illustrates, however, several error-processing techniques you may find useful in your own programs. It also shows another way to use the keyboard buffer to enable a program to jump into immediate mode, give itself a 'goto', and resume operation.

One of the greatest strengths of Waterloo microBASIC is its excellent error processing, akin to that on many time-sharing versions of BASIC. Errors are handled by invoking either the 'on [type of error] ignore' statement or the more powerful and flexible statement shown left, below. Both must be placed in program at a point prior to the place where a specific type of error might occur. The structure shown on the left allows sophisticated error handling; we are not limited as to the amount of code between 'on...endon'. If the particular error can be handled at all, we may resume program after the error either on the line where the error occured or on the next line ['resume' or 'resume next' statements]. We may restore normal system error processing by an 'on [type of error] system' statement after we pass the trouble spot, or [an important point] by a 'stop' statement.

```
10 on [type of error]
20    [error processing instructions]
30    resume [or resume next]
40 endon
```

Chapter 12 of the microBASIC manual lists the various types of errors which can be processed. Though the manual does not say so, we may not employ the general statement 'on err', which is refused. Among the most useful of the statements is 'on attn'; if invoked, it keeps the STOP key from interrupting a program, and it also causes the STOP key to be ignored during 'input' and 'get' statements.

Curiously, and rather unfortunately, the STOP key is not defeated by 'on attn' for 'linput' statements. If we try to write bulletproof programs, we may not use 'linput' for keyboard input, nor may we read random characters [including STOP, chr$(3)] from disk files using 'linput #'.

```
185 CS$=chr$(12) : CR$=chr$(13) : D$=chr$(10) : H$=chr$(1)
190 ! If system variable date$ is set then listing will be dated.
195 ! WARNING: Change line numbers or renumber this program at your own risk!
200 ! Lines marked by ##### must be altered by hand if program is renumbered.
205
210 on attn                              ! Process STOP key up to line 370
215    resume
220 endon
```

Lines 210-220, above, show the simplest way to process the STOP key. If that key is pressed, the 'on attn' causes the program to 'resume' on the next line. Note well that mBASIC processes STOP key errors in a special way; it waits until the end of the current line to check for STOP in the keyboard buffer. Hence 'resume' jumps to the next line; 'resume next' skips a line. 'Resume' is thus identical in effect to an 'on attn ignore' (which is not allowed).

MicroBASIC processes all other types of error differently; it transfers control of program to the 'on...endon' structure as soon as an error is detected in the current line. If we employ 'resume', the same line is reprocessed; 'resume next' goes to the next line.

```
225 on conv                            ! Process non-numeric input @ line 290
230    ce%=1 : resume next
235 endon
240 on ioerr                           ! Trap or process disk drive error.
245    print CS$ : c%=cursor(826)
250    print 'PLACE DESIRED DISK IN DRIVE #';dr% : c%=cursor(c%+150)
255    print 'Check for Write-Protect Tab. Press RETURN when ready.' : dio%=1
260    get a : if a<>13 then 260
265    resume next
270 endon
275 CS$=chr$(12) : CR$=chr$(13) : D$=chr$(10)
280 get a$ : if a$<>'' then 280        ! Clear keyboard buffer
285 print CS$ : if cursor(830) then input "Which disk drive? ",dr$
290 dr%=value(dr$)                     ! String to integer conversion.
295 if ce% then ce%=0 : goto 285       ! Conversion error detected.
300 if dr%<0 or dr%>1 then 285         ! Only drives 0 and 1 exist.
305 open #121,'disk/'+value$(dr%)+'.zz', output    ! Test for disk in drive.
310 if dio% then dio%=0 : goto 305     ! Disk I/O error detected.
315 on conv system                     ! Reinstate system error processing.
320 poke 297,0                         ! Disable keyboard.
325 close #121
330 scratch 'disk/'+value$(dr%)+ '.zz'  ! Scratch test file
335 on ioerr system                    ! Reinstate normal I/O processing
340 ncr%=1 : d$="di'disk/"+value$(dr%)+"'" : al$=d$ : jj% = -1
345 open #131, 'keyboard', inout       ! Allow read & write on keyboard buffer
350
355 ! >>> Returns here to continue displaying directory on screen.
360 print #131, CS$;d$;rpt$(CR$,ncr%);'goto 375';CR$    ! #####
365 stop                        ! Dumps keyboard buffer filled @ line 360
370
375 !>>> Program continues to run starting here after keyboard dump.
380 open #131,'keyboard', inout       ! Reopen keyboard after 'stop' statement
385 on attn                           ! Reset STOP key processing after going
390    resume                         ! into simulated immediate mode @ 365
395 endon
```

When we execute a STOP at line 365 and go into immediate mode, we restore normal error processing. As we resume program, we must repeat the 'on attn' sequence in lines 385-395. Note that 'on attn' does not keep a user from putting STOP into the keyboard buffer; it is simply a method of ignoring STOP. Should the buffer contain a STOP when we enter a program, STOP will be processed before any of our program is executed (our program is STOPped). We do not face this problem in any ordinary programs which are 'run' from immediate mode, for if we press STOP, the program STOPs, and we see 'Ready'.

In a program such as this, however, we jump in and out of immediate mode. If we press STOP at just the right moment, we may slip it into the keyboard buffer and

STOP our program before we reset 'on attn' at line 385. This raises a general question: how do we best keep a user from entering unwanted characters and commands in critical parts of a program? One solution: disable the keyboard with 'poke 297, 0', which leaves the keyboard utterly dead (we must avoid this when in immediate mode, for we lose all communication with the computer and must recycle). We may re-enable the keyboard with 'poke 297,255'. We employ the method several times in PRINT DIRECTORY.

Note that a program 'stop' statement does not re-enable the keyboard; if at line 370 or immediately thereafter you press the STOP key, no STOPs are sent to the buffer. When it is time to accept further keyed input and 'on attn' has been set safely, the keyboard is temporarily re-enabled in line 470; when a character has been received at line 475, we again disable it until at end of program we bring it back to life. Note further that when we disable the keyboard itself, we may still use the buffer to receive and transmit data.

Be extremely careful in disabling the keyboard; if your program bombs, you have no way to recover. Debug your program totally before you add STOP key processing or disable the keyboard. You must, in addition, plan to handle other errors with no resort to the keyboard, as we do in this program.

[For lack of space, we must continue this article next issue. The remainder of the program follows. Comment out all 'on attn' lines and all pokes which disable or enable the keyboard when you enter the program and keep these lines dead until it's on disk and running well. Many of the features will become clear in the next installment. Ed.]

```
400 aa%=-79
405 for ii%=1 to 24                  ! Reads screen lines into AA$
410    print #131, CR$:aa%=cursor(aa%+80) : linput "",aa$
415    if aa$=d$ then 440            ! Skip if d$="di'disk/dr#'"
420    if idx(AA$,aa$) then 440      ! Skip if already present
425    if aa$='goto 375' then 445         ! #####
430    if aa$(len(aa$)-6:len(aa$))= 'S FREE.' then 450 ! Quit on blocks free.
435    jj%=jj%+1 : AA$=AA$+aa$       ! Add DIR entry to AA$
440 next ii%
445 ncr%=ncr%+1 : goto 355           ! Increase # of dir pages and rescan.
450 BF$=aa$ : close #131              ! BF$ = number of blocks free on disk.
455
460 print CS$ : if cursor(830) then print "SET PRINTER PAGE TOP"
465 print D$;D$;D$;tab (29);'[PRESS ANY KEY WHEN READY]'
470 poke 297,255                      ! Reenable keyboard for line 475
475 get a$ : if a$='' then 475        ! STOP key still ignored.
480 poke 297,0
485 open #141, 'ieee4', output        ! Set up Comm 8023 printer (132 col)
490 AT$=AA$(4:21): if date$<>'' then AT$=AT$+ " ["+date$+"]"
495 AA$=AA$(28:len(AA$)) : lA%=len(AA$) ! Trim disk header.
500 os%=0 : il%=jj% : if jj%>120 then il%=120
505 tbc%=((il%>60)+1)*16               ! Spaces to center title line.
510 print #141, tab (tbc%-14);H$;'DISK DIRECTORY'
515 print #141, tab (tbc%-len(AT$)/2);AT$;D$  ! Disk title line.
520 for kk%=1 to 60                   ! Print out AA$ (1 or 2 columns)
525   for ll%=kk%+os% to il% step 60
530     ff%=27*(ll%-1)+1
535     if ff%<lA% then print #141, AA$(ff%:ff%+26),
540   next ll%
```

```
545   print #141
550 next kk%
555 if jj%>120 and os%=0
560   os%=120 : il%=jj% : print #141, rpt$(CR$,2)
565   print #141, "DISK DIRECTORY (cont)"
570   print #141, AT$;D$ : goto 520
575 endif
580 print #141, tab (tbc%-15);BF$;"    ";jj%;"FILES.";D$;D$
585 close #141                         ! Prints just 66 lines for paging
590 poke 297,255 : on attn system      ! Restore keyboard, STOP processing
595 end                                ! Copyright 1983, John A. Spencer
```
****

THEM DURN SWITCHES   Early SuperPETs were sold with two switches on the right side of the case: one makes ROM out of RAM by write-protecting the upper 64K of memory; the second selects the processor used (6502 or 6809). What follows is pieced together from reports by Richard Schumacher of St. Louis and from Associate Editor Terry Peterson (ye ed has two switches on SPET No. 238): You can't put EPROMs in two-switch SPETs; the circuits which disable the 6502 ROMs when you switch to 6809 mode will not disable EPROMs. If EPROMs are socketed at UD11 or UD12, you can't run 6809 mode (without extra switches).

Two new switches solve the problem. Switch three controls UD12 ($9000-$9fff). If the switch is on, the (EP)ROM in that socket is active, and the bank-switched upper 64K of memory in SPET is disconnected. Since any ROM at $9000 can be unloaded to disk (see Roy Busdiecker, Vol. 1, pp. 28-29) and then reloaded from disk into any of the 16 4K banks of switched memory, and any of those banks are accessible from 6502 mode, UD12 isn't essential unless you like ROMs. You turn switch three OFF for 6809 mode.

Switch four controls socket UD11 ($a000-$afff), where WordPro and WordCraft ROMs (EPROMs?) usually reside. If you use this socket, turn the switch on in 6502 mode, and off for 6809 mode.

Walt Kutz has informed us, and COMMODORE magazine has said that the switches are free, but that installation by a dealer is not. Because the switches are inexpensive and free, we have a lot of letters saying dealers don't have them. We rather doubt the distributors are trying very hard. Suggest the Slobbovian Ultimatim: Get the damn switches or I take my business elsewhere. Where is elsewhere? Try Fisher Scientific (last issue). Aren't alternative sources handy?

Early on, the switches on our SPET fell off or got flipped (usually while in program!) when we dusted or chased a pencil hiding under SPET. Once we yanked a handbook out, and extracted a switch, wire entrails and all, from its socket inside SPET. Enough! So we got a tube of our favorite cement (DURO Rubber), sanded through the paint, coated two spots on SPET and the mounts of both switches, let 'em dry for 15 minutes, and pressed 'em on--with the switch toggles facing the operator, not the desktop. Now we can read the legends on the switches; no pencil or book or dustcloth flips ary a toggle--a distinct improvement.
****

SUPERPRINT FOR THE SUPERPET [PRINT ON INTERRUPT WITH PIRQ]

by Gary L. Ratliff, PO Box 829, Sanatorium, Mississippi 39112

Last issue, we presented a 6809 screen dump and promised to find some additional uses for the technique. This issue, we keep that promise and show how the routine serves as the base for an interrupt-driven process, which works with all printers; you may, however, need to add a line feed (not hard; take out some 'comment' semicolons in the .asm listing) if your printer fails to perform a line feed with each carriage return ($0d in assembler). Not only does this program work on all printers, it works in all languages and facilities in SuperPET including the 6502 Development System. Hence, whenever you run 6809 code, this program dumps the screen to printer upon demand (Yes, in FORTRAN, PASCAL, COBOL, APL, mBASIC, the microEDITOR, the MONITOR, and all DEVELOPMENT facilities.)

We call it 'pirq' since it 'prints on interrupt'. You load it from the main language menu in less than two seconds with: pirq <RETURN>. It occupies only 186 bytes at the very top of user memory, and sets end-of-memory itself. The version published dumps whenever you hold the backslash '\' key down and let it repeat forty times. It stops the dump on the row of backslashes, and then erases them. You can substitute any other key for the 'trigger' if you care to, and want to use the backslash in text.

In APL, the left-arrow key is probably the best candidate (ordinal 95, or $5f). The REPEAT key (ordinal 127, or $7f) is also available, if otherwise not used. The backslash '\'is handy for text like this. It's easy to substitute the key you want. You will still require at least two backslashes at left margin to stop the dump short of 24 lines. Once you've loaded PIRQ from the main language disk, you are returned to the language menu; load anything you want. PIRQ remains in memory so long as you stay in 6809 mode; you can switch languages or facilities.

The history of PIRQ shows what collaboration in SPUG can do. Jeff Larson came up with a printer routine which gently line-feeds all blank lines, instead of shovin the printhead from margin to margin. His routine also returns the carriage to left margin as soon as the last character on a line is printed. All this produkes a very swift but gentle printout. Jeff sent his new version to Dick Barnes, while I sent Dick an early version of PIRQ. Dick said: 'Hey, Gary. These two are great. Why don't you marry them?' PIRQ is the result.

In Aug.'82, COMPUTE! published a summary of KEYPRINT programs for most Commodore machines. KEYPRINT allows people using the 6502 to hit the "\" key (one touch!) to dump the contents of the screen to a printer. (Had we been in 6502 mode, we would have dumped.) Each different machine needs a different version of the 14 published by Brannon last August. Some users would like to 'trigger' the dump with a different key. KEYPRINT, in 6502, always dumps the full screen (whether blank or full), and runs the printhead from margin to margin on all lines. Being aware of these problems, we improved upon KEYPRINT with PIRQ. (Those who want to use KEYPRINT in 6502 mode in SPET should enter version 11.c.(80D), as published in the Aug. '82 COMPUTE!, starting at p. 103.)

The original version of KEYPRINT appeared in COMPUTE! No. 7, Nov/Dec '80. I have (sob) every issue of COMPUTE! except No. 7. If any of you have an extra copy of issue 7, I'll swap you a No. 5 for it, so both our collections are complete. [If you can help Gary, please do. He is a major contributor to the Gazette and SPUG; we all owe him a great deal. Find his address on the title line, above. Ed.]

This program also illustrates several programming techniques. (This is not in my column BITS & BYTES and will show that I don't need an extra month to find my coding errors). First, the program shows how to use an interrupt; we attach it to the system via the routine conbint_. We remove from user memory the amount used by PIRQ by setting memend_. Service_ is set to zero so that after PIRQ is loaded, system control will pass back to the main menu. The original contents of the IRQ vector need to be tucked away unless you plan to completely redo the entire interrupt system. Second, we employ a technique to control general purpose routines. Any routine to be run by the system should not change the contents of the stack pointer. The status of other interrupts and system tasks would then be lost by this process. We may use neither SWI nor RTI; instead, we reenter the system handler via a RTS instruction, which issues the RTI instruction. Third, we place the finished load module on the system disk (language disk); we load it from that disk with any short name (PIRQ loads with: pirq <RETURN>).

Fourth, and probably most important, we have created a technique for a standard set of 'user library' routines, to which we can all refer without re-defining them each time we write a new program. Jeff Larson's memory map gives us access to a number of routines that were not included in the Waterloo export files on the language disk. Since Waterloo has two libraries of such routines on the language disk, we created a third: the user library, or 'usrlib'. If all of us use the standard names for the routines set down in the memory map published in this issue, these routines (where they do not duplicate Waterloo library exports) can then become a part of a growing 'userlib', which, like the Waterloo library, is put on and kept on the language disk.

We have the start of such a library, and use it to link PIRQ. As your first step in entering PIRQ, enter the microEDITOR, type in 'usrlib.exp', and then file it on the language disk as 'usrlib.exp'. The linker will then use the memory locations in that library to link the routines we need.

After you have assembled and linked PIRQ, you will have a load module filenamed 'pirq.mod'. Put the disk containing the module in drive 1; do a cold start in 6809, and load it from menu with 'pirq.mod <RETURN>'. If it tests okay, put the system disk in drive 1, and the 'mod' disk in drive 0. Copy 'pirq.mod' to your language disk with Jim Swift's method in the microEDITOR in Development, (left).

```
g ieee8-15.C1:pirq=0:pirq.mod
```

From this time forward, PIRQ can be loaded whenever you enter 6809 mode. Remember it stays in memory until you leave 6809, and that the top of user memory is automatically set at $7f00.

If you have a serial printer, you'll have to call 'setup' to establish the baud rate before you load PIRQ. At any time you want the screen printed, just move the cursor to the very left margin of the line on which you want printing to end and hold down the slashbar key until the printer kicks off. If you want, you may print only two slashbars, and then hold down the spacebar (or TAB or SHIFT keys) until the keyboard buffer fills. When the dump finishes, the garbage slashbars are automatically erased. If you use another key as 'trigger', and want to dump the whole screen, just leave the slashbars out.

While PIRQ works with all languages, special measures are needed in APL. (1) If you use a CBM or ASCII printer, you print out keys as they appear on the ASCII keyboard; viz., the apostrophe "'" in APL prints as K (shifted K in APL is '). (2) Workspaces created prior to PIRQ will produce a WS NOT COMPATIBLE message when you attempt a)LOAD. For a workspace named 'TEXT', we can cure the problem: 1. Clear the workspace)CLEAR. 2. Copy the workspace into memory:)PCOPY TEXT. 3. Reestablish the name of the workspace:)WSID TEXT. 4. Save the converted WS with)SAVE TEXT. End of example. (3) In APL, the cursor normally returns to five spaces into a line. You must, therefore, backspace to the left margin before you drop your reverse slashes to activate PIRQ. Last, PIRQ will not print overstruck characters to printers which require a backspace for such characters.

Following are the assembler and linker files for PIRQ. You will see **** where you may change the 'trigger' key and where you may add a linefeed for printers which do not generate one with a carriage return. Pick your printer type at the start of the program: 1 for serial, 2 for commodore, or 3 for ieee4. And remember to put 'usrlib.exp' on the language disk!

```
export membeg_  = $20   ; supplied 6809 asm documents from Waterloo
export memend_  = $22   ; "
export service_ = $32   ; "          [This is file usrlib.exp]
export intvctr_ = $0100 ; "
export bufpnt_  = $012e ; discovered 2/83 by jeff larson

    ; pirq.asm--the file for the assembler
    ; a routine to process print dump as interrupt
    ; by gary l. ratliff  : title 'pirq'
    ; uses screen print routine written by jeff larson with modifications
    ; needed to have routine operate as an interrupt
        xref    memend_
        xref    service_
        xdef    main
        xref    conbint_
        xref    intvctr_
        xref    openf_
        xref    closef_
        xref    fputchar_
        xref    bufpnt_
        xdef    start
        xdef    print
        xdef    resvd
        xdef    outpt

type    equ     3        ; user equates for printer: 1=serial, 2=printer
size    equ     80       ; 3=ieee4 : equates attach to irq handler
quit    equ     0
main    equ     *        ; this will be program origin ($7f00)

        ldd     # main   ; removes one page from memory
        std     memend_  ; and sets to return to main menu
        ldb     # quit
        stb     service_

        ldx     # 8          ; saves original address used by
        ldd     intvctr_,x   ; normal interrupts
        std     resvd

        pshs    x            ; establishes start as the address
        ldd     # start      ; to service the irq type interrupts
        jsr     conbint_
        leas    2,s
        rts

start   jsr     [resvd] ; first service normal interrupts
        ldb     [bufpnt_]
        cmpb    # '\     ; *** test for reverse slashbar. Change to ordinal of
        beq     doit     ; desired trigger key: left-arrow=$5f, repeat=$7f, etc.
                         ; remove apostrophe if a new trigger key is used.
        rts              ; not found so exit irq routine
doit    sei              ; disable further interrupts
```

```
        jsr     fill     ; fill buffer to prevent this
                         ; condition from being recognized
                         ; again during service
        ldd     # mode
        pshs    d
        ifeq    (type - 1)
         ldd    # typ1   ; serial type printer
        endc
        ifeq    (type - 2)
         ldd    # typ2   ; printer (Commodore 4022 etc. printers)
        endc
        ifeq    (type -3)
         ldd    # typ3   ; ieee4 devices which use true ASCII code
        endc
        jsr     openf_   ; open the printer file for processing
        leas    2,s      ; clear parameters from stack
        std     outpt

        if ne            ; process if file opened correctly
         ldd    #$8000 ; address of screen start
         std    line
         loop
          ldx   line
          jsr   print
          ldx   line
          ldb   # 80
          abx
          stx   line
          ldd   ,x
          cmpd  #$5c5c   ; test for \\ at line start
                         ; if found the printing is to stop
          quif eq
          cmpx  #$8780   ; also routine is done when
                         ; 24 lines have been printed
        until eq

        tfr     x,d
        subd    #$8000
        std     $0122    ; this portion sets cursor position to
                         ; the line containing the \\\\ 's
        jsr     $db30    ; this sends EOL sequence to erase
                         ; the slash characters
        ldd     outpt
        jsr     closef_
        endif
        cli              ; enable interrupts
        rts              ; task complete
print   tfr     x,d      ; prepare to adjust
        addd   #$50      ; scan line backwards
        tfr     d,y      ; to only print characters
        loop
         ldb ,-y         ; skipping all trailing blanks
         cmpb #$20       ; similar to -trailing
         quif ne         ; in FORTH
         cmpy line
         quif eq
        endloop
        tfr y,d         ; now make adjustments to
        subd line       ; to the line size to
        addd #$01       ; be printed
        tfr d,y
        loop
         ldb    ,x+
         pshs   y,x,d
         ldd    outpt
         jsr    fputchar_
         leas   2,s
         puls   y,x
         leay   -1,y
        until eq
        ldb     #$0d
        pshs    d
        ldd     outpt
        jsr     fputchar_

        puls    d
; use the following lines if you need a line feed with each CR ****
       ;ldb #$0a          ; the line feed character
       ;pshs d
       ;ldd outpt
       ;jsr fputchar_
       ;puls d
; by stripping off the semicolons in the lines just above ****
        rts

fill    ldx     #$130    ; fill key buffer with spaces
        ldb     #'
again   stb     ,x+
        cmpx    #$158
        bne     again
        rts

                        ; data area
typ1    fcc     "serial"
        fcb     0
typ2    fcc     "printer"
        fcb     0
typ3    fcc     "ieee4"
        fcb     0
mode    fcc     "w"
        fcb     0
outpt   rmb     2
line    rmb     2
resvd   rmb     2
        end             ; End of pirq.asm
```

*   *   *

```
"pirq"
org     $7f00                  [This is file pirq.cmd for the linker.   ]
include "disk/1.usrlib.exp"    [Remove these two comments before filing.]
include "disk/1.watlib.exp"
"pirq.b09"
```

****

RELATIVE FILES - by Loch Rose, 102 Fresh Pond Pkwy, Cambridge, MA 02138

****

A relative file is a data file that is divided into discrete records, each record containing one or more pieces of data. Each record is numbered, so you can directly write to or read from any individual record in a relative file. This can really speed up data access: for instance, I store every day's Value Line stock index future closing prices in a different record. When I need a given day's prices, I can pull them directly out of the appropriate record.

A relative file is created the first time that you OPEN a new file with a name of the form "(f:XX)disk/0.filename,rel". [Note: mBASIC commands will be capitalized in the text, for clarity, but should be entered in lower case in a program.] "filename" is a 1-16 character file name, "disk/0." is optional (see p.30 in SPET System Overview), and "XX" is the record length (1-254) in bytes. The record length cannot be altered once the file has been created, and all records in a file are the same length.

This demonstration program creates a relative file with 20 records, numbered 0 to 19, record length 80 bytes. Into record #0 we'll put the number 0, a comma, the string "0", a comma, and the string "Record #0"; and so on for the other records.

```
10 c$=","
20 open #2,"(f:80)number, rel", inout    ! 'inout' allows read & write
30 for i=0 to 19
40   print#2,rec=i,i;c$;value$(i);c$;"Record #"+value$(i) ! write to record 'i'
50 next i
60 input #2, rec=5, n, num$, rec$        ! read contents of record #5
70 print n, num$, rec$
80 close #2 : stop                       ! I suggest STOREing this program
```

The "rec=i" in line 40 directs the data to record "i", just as "rec=5" reads data from record 5 in line 60.

How you structure the data within a record is up to you, but the primary consideration should be ease of input. I recommend your separate data items by comma characters, as I have done, so you can use the INPUT# statement. Using the <RETURN> character (chr$(13)) is not as good, as it interferes with LINPUT#.

Relative files reserve space on disk for all records from record #0 up to the highest record to which you have written. For example, watch the error indicator as you run the following:

```
10 open #2,"(f:200)demo,rel",output       ! note that 'output' can be used
20 recnum=25
30 print #2, rec=recnum, "record"; recnum  ! write to record #recnum
40 close #2 : stop
```

Check the number of blocks used by the file via the directory command (it should be 22 blocks). Then substitute 50 for 25 in line 20 and run it again; there should now be nearly twice as many blocks. Even though you didn't use any records 26-49, space was reserved for them; I suggest putting a filler character, such as chr$(255), into unused records like these in event you access them by accident. If you watched the error indicator, it should have flickered red. This is normal whenever you expand a relative file, so ignore it.

You lose nothing in using relative files: you can treat them like a sequential file, if you want. This program runs through the data file until it finds the string "10":

```
10 open #2,"(f:80)number, rel", input  ! use same name; 'input' not 'inout'
20 loop
30    input #2, a$                     ! note omission of 'rec='
40    print a$
50 until a$="10"
60 print "We found ";a$;"!!!"
70 close #2 : stop
```

You can shortcut the procedure by inserting "15 input#2,rec=5,a$,a$,a$", which positions you after record 5 in the data file. GET# and and INPUT# statements without a "rec=" clause operate at the point within the data file where the last operation left you, just as with sequential files.

I would avoid using GET# statements with relative files, as LINPUT# is a faster and more convenient way of reading the entire contents of a record. The state- "100 linput#2, rec=10, a$" stores in a$ the contents of record #10, or the contents up to the first chr$(13) character (if any).

Notice that while all data were INPUT as strings in the above program, the first datum in each record was a numeric variable, "i" (see first program). This works because all data are stored on disk as strings; when reading the data, you you may choose to INPUT numeric variables as such or as strings.

You can modify a relative file easily, provided that you remember that any writing to a record wipes out ALL its former contents. For example, suppose we alter the second datum in record #5:

```
10 c$="," : newdata$="505"
20 open #2,"(f:80)number,rel",inout
30 input #2, rec=5, n, num$, rec$             ! read current record #5 content
40 print #2, rec=5, n; c$; newdata$; c$; rec$  ! rewrite record
50 input #2, rec=5, nextn, nextnum$, nextrec$  ! check new contents
60 print n, num$, rec$ : print nextn, nextnum$, nextrec$
70 close #2 : stop
```

Relative files waste space on disk compared to sequential files because all records must be as long as the record with the lengthiest data, and so most records will contain some unused space. Also, the DOS limits you to 720 records per file, though new 8050's are supposedly not so limited. But the speed and ease of access mean that I use relative files whenever possible, whenever data break down naturally into records of 254 bytes or less.
****

BITS BYTES & BUGS For lack of space, we will carry Gary Ratliff's regular column next issue (you have enough assembly language material this issue to last a while!). Gary says the error in the last BITS & BYTES column is one that BASIC programmers make less frequently than others, as BASIC, unlike many high-level languages, sets all variables to zero before a run. Last issue, he defined the screen as equal to $8000, but never stuffed that value into the y register, either before or after the ldx # msg. The code assembles okay but if executed wipes out the operating system, since index register y contains 0000 and the message it points to overwrites vital pointers on the zero page, starting at the very first address (0000!). Gary, you are MEAN!!!!

****

SUPERPET MEMORY MAP

by Gary L. Ratliff

The companion article to this one is PIRQ, an interrupt-driven routine for printer which dumps the screen. If you examine the code, you will notice that we use some routines which are not documented. For the past several months, various members of SPUG have exchanged data on the routines they have found. Much of the information is contained on the files watlib.exp and fpplib.exp, found on the system disk. If you haven't already done so, print hard copy of these files. With this information in hand, we (all who cooperated) traced the actual location of these routines in the system ROMs. Jeff Larson did most of this chore; his information was verified by me and also by Reg Beck; with three of us at it, we caught most errors in locations of text strings. Not all of the Znd (See beow. Ed.) have been tested, however, as they are a very recent addition to the material in this memory map. And we did find a number of useful but previously undocumented routines.

This map shows the current state of all memory locations so far identified. The honor of naming the routines and locations belongs to the person making the discovery. However, to meet a publication deadline Dick sugested that I temporarily name routines discovered by Jeff and others; such routines end with *, and are subject to renaming by their discoverer. When the information is comlete, SPUG will publish a file called 'usrlib.exp' (On disk. Ed.) which will include all known locations and routines not now included in the Waterloo libraries. This user library may then be included in any linker file by a simple, short reference in any .cmd file to "include 'disk/1.usrlib.exp'". [A stroke of genius by Gary; we now have a simple, standard way to identify and use the undocumented routines throughout SPUG. Ed.]

Some old PET hands remember how ROMS changed in early PETs, and how the locations we had so laboriously discovered were instantly outdated when the ROMs were updated--amidst great cries of woe and pain. The ROM updates were worth the trouble. We hope Waterloo updates ROMs for SuperPET; if Waterloo should, we have a microEDITOR and can simply update our 'usrlib' file to conform to any changes. A word of caution, though: if you write a lot of programs in which you address current data locations, rather than using a library routine, any update might leave you with the task of rewriting every address in your programs, instead of simply using the new 'usrlib' file to update them automatically. Example: we now know where 'date$' is kept in the zero page; an update can easily change its location, but the library routine (if changed) will automatically go to that new location without you having to bother with a change in address. My advice: stick with the library routines where you can use them.

Now we present the first published SUPERPET MEMORY MAP.

loc.	NAME	Description	loc.	NAME	Description
0020	MemBeg_	Start Usr Memory	0022	MemEnd_	End Usr Memory
0032	Service_	0 rts menu; reruns	0040	Free1	Unused. 0040_0060
0080	FpWk	Float area to 0097	0080	fac1	Fp acum 1 see 6809 doc.
00x0	(Sign)	Fac 00 pos ff neg	00x1	(Exp)	Exp. in excess 128 form
00x2	(MSB)	Normalized fraction	00x3	(Fract)	Fraction part 1
00x4	(Fract)	Fraction part 2	00x5	(Fract)	Fraction part 3
00x6	(Fract)	Fraction part 4	0088	Tmpwk	Fltng point work area
0090	fac2	Fp acum 2 to 0097	0098	Free2	Unused zpage to 00ff
0100	IntVctr_	Sft cpy irq vectors	0102	SfSWI3	Soft copy SWI3
0104	SfSWI2	Soft copy SWI2	0106	SfFIRQ	Soft copy FIRQ
0108	SfIRQ	Soft copy IRQ	010a	SfSWI	Soft copy SWI

010c	SfNMI	Soft copy NMI	010e	Tab1	1st tab setting
0110	Tab2	2nd tab setting	0112	Tab3	3rd
0114	Tab4	4th	0116	Tab5	5th
0118	Tab6	6th	011a	Tab7	7th
011c	Tab8	8th	011e	Tab9	9th
0120	Tab10	10th	0122	Curpos*	Cursor pos - 0123
012c	Kyptr1*	Pointer1 to kybuffer	012e	Kyptr2*	Pointer 2 to kybuffer
0130	Kybuf*	Kybuffer (0130_0157)	0160	Time*	Holds time to 0163
0160	Hours	Time hours 00-17	0161	Mins	Time minutes 00-3b
0162	Secs	Time seconds 00-3b	0163	Jiffs	Time jiffies 00-3b
0164	Date*	Date string to 016e	0182	PCSav	Saves Monitor PC reg.
0184	ASav	Saves Monitor A reg.	0185	BSav	Saves Monitor B reg.
0186	XSav	Saves Monitor X reg.	0188	YSav	Saves Monitor Y reg.
018a	USav	Saves Monitor U reg.	018c	SSav	Saves Monitor SP.
018e	CCSav	Saves Monitor CC reg.	018f	DPSav	Saves Monitor DP reg.
01f8	StkBot	Bottom sys stack	0220	SysStk	Normal System stack
0220	CrBank	Current bank used	0223	UsVctr	User crtd. irq vectors
0226	UsSWI3	Usr vctr for SWI3	0229	UsSWI2	Usr vctr for SWI2
022c	UsFIRQ	Usr vctr FIRQ	022f	UsIRQ	Usr vctr IRQ
0232	UsSWI	Usr vctr SWI	0235	UsNMI	Usr vctr NMI
0400	Linbuf*	Text buffer to 0450	0600	LnStk	Language Stacks to 09ff
0a00	UsWkSp	User workspace-7fff	8000	Screen	The video scn. to 87cf

87d0 Hyde "Hidden" screen memory reported by associate editor Roy Busdiecker in article which first appeared in COMPUTE # 7. Goes to 87ff. The format of fac2 is identical to that of fac1. The generalized decription is given using offsets from start address. The Usr irq vectors are set up through Conbint_; form is bank# address.

Math routines from fpplib.exp JUMP TABLE: A000_A06E compiled by Jeff Larson. My explanations are terse; a full description is found in the file fpplib.exp, which is on the system disk.

a06f	cnvf2s_	cvrt flt 2 string	a1b7	cnvs2f_	cvrt string 2 float
a2bc	fload_	load fac1	a2db	fload2_	load fac2
a2fa	fstore_	mv fac1 2 memory	a314	cnvfi_	fac1 2 integer
a33f	fraction_	frac part of fac1	a35d	exponent_	expnt of fac1
a36b	fmul10_	fac1 x 10	a386	trf1f2_	transfer fac1 2 fac2
a39a	cnvif_	cvrt integer 2 float	a3b0	fzero_	fac1=0
a3e4	fcmp_	compare fac1 2 fac2	a3e7	fcompare_	not mentioned in doc.
a41a	ftest_	test fac1	a42b	fsub2_	ld fac2 sub from fac1
a430	fsub_	fac1 minus fac2	a434	fadd2_	ld fac2 add to fac1
a439	faddhalf_	fac1 plus 0.5	a43f	fadd_	fac1 plus fac2
a4b6	shiftl_	not mentioned in doc.	a4c3	fmulby_	ld fac2 mul by fac1
a4c8	fmul_	fac1 times fac2	a54b	fdivby_	ld fac2 div into fac1
a555	fdiv_	divide fac1 by fac2	a5e6	fneg_	negate fac1
a613	addacc_	not mentioned in doc.	a65d	sqr_	square root of fac1
a666	power_	fac1 raised to fac2	a6e2	exp_	e raised to pwr fac1
a739	log_	log base e of fac1	a799	atn_	arctangent of fac1
a7d7	cos_	cosine of fac1	a7dd	sin_	sine of fac1
a939	ffloor_	lgst intger less fac1	a93e	finteger_	not mentioned in doc.

Consult the document entitled 'Waterloo 6809 Assembler: Waterloo microSystems SuperPET Specifics', available from Jennifer Uttley, Editor, WATNEWS, University of Waterloo, Computer Systems Group, Waterloo, Ontario, Canada N2L 3G1, which will explain the parameters and use of these routines. The issue at hand is dated September, 1982; later version may even explain the routines not covered in my copy of the reference. Next, a few general items:

aa27	Mnwds1*	Menu words (to aa5e)	abf2	Sldsk*	select %n%n disk/1 msg
ad0e	Pgntfd*	Pgm not found msg	af67	Mnstup*	Setup menu

Library routines from watlib.exp compiled by Jeff Larson. Consult the Assembler Manual for how to use these routines. JMP TBL: B000_B11a.

addr.	NAME	addr.	NAME	addr.	NAME	addr.	NAME	addr.	NAME
b11a	IntPow10_	b124	DevList_	b1c1	InitSrd_	b1e5	GetChar_	b1ea	PutChar_
b1f9	PutNL_	b1fe	GetRec_	b20f	PutRec_	b221	Printf_	b23b	Openf_
b292	Closef_	b2a6	FGetChar_	b2fd	FPutChar_	b324	FPutNL_	b333	FGetRec_
b367	FPutRec_	b3aa	FPrintf_	b4a3	FSeek_	b4c9	Eor_	b4d3	Eof_
b4e5	Errorf_	b510	ErrorMsg_	b518	Scratchf_	b543	Renamef_	b58b	Mount_
b5bd	TimeOut_	b5d2	DirOpenf_	b610	DirReadf_	b627	DirClose_	b66a	Iotime*
b677	___RET	b67f	___RET2	b687	___MUL	b6c4	CARRYSET_	b6ce	___DIV
b6d8	___MOD	b742	___NEG	b748	_RSHIFT	b74a	_LSHIFT	b782	StrEq_
b7b1	Length_	b7ce	Equal_	b7ff	CopyStr_	b80a	PrefixSt_	b832	SuffixSt_
b84a	Decimal_	b87e	StoI_	b8e6	BtoHS_	b946	HStoB_	b9c7	Hxstng*
b9d8	IsAlpha_	b9ee	IsLower_	b9fa	IsUpper_	ba06	IsDigit_	ba14	IsDelim_
ba2d	IsHex_	ba58	Hex_	ba77	ZUpStr_	ba8d	Upper_	ba9e	ZLoStr_
bab4	Lower_	bacb	ItoS_	bb48	ItoHS_	bb68	Copy_	bba7	TableLoo_
bbf1	BankSW	bc21	BankInit	bc2d	ConBInt	bc45	UIntHdl	bc67	UIntTab
bc75	Spawn_	bcab	Suicide_	c1c8	Dvntprs*	c1e1	Request_	c1f5	SysIOIni_
c2d8	SysRead_	c52c	SysWrite_	c6c6	SysNL_	ca57	Dskerr*	cbca	Albet*
d27b	Uhxsng*	d4be	Nomem*	d4cc	TIOInit_	d4d2	TGetChr_	d4f0	TPutChr_
d500	TGetCurs_	d509	TPutCurs_	d518	TBreak_	d521	TSetChar_	d55c	SIOInit_
d598	SPutChr_	d5bb	SGetChr_	d62e	SBreak_	d714	Bputscn*	d721	Zndrgt*
d729	Zndup*	d743	Zndlft*	d74b	Zndhm*	d750	Zndr/s*	d751	Zndasc*
d7e3	TabSet_	d817	TabGet_	d872	Zndcr*	d887	Znddn*	d945	ZndTab*
d9a9	Zndclr*	d9de	Zndins*	da53	Znddel*	db30	Zndeol*	dc4a	ChrPtr*
dd82	KyputB*	de01	KBEnable_	de07	KBDisabl_	decd	Scroll*	e0f4	SetTime_
e107	SetDate_	e13c	GetTime_	e158	GetDate_	e15f	PassThru_	e1c7	FxChBuf*

An explanation of some of the newly-discovered routines is in order: Mnwds1 is a list of the main menu words, viz., setup, apl, etc. Sldsk is the "select disk " message. Pgntfd is the message "loading ... program not found." Mnstup is the setup menu for configuring the serial port. Iotime is the I/O time out message. Hxstng is the string "0123456....def." Dvntprs is the message "Device not present." Dskerr is an assortment of disk error messages. Albet is a subset of the alphabet from A to P. Uhxsng is the same hex string using uppercase letters 'ABC...', etc. Nomem is the message "out of memory." Bputscn puts the character in the b register on the screen and advances the cursor. Znd is a family of routines which use Bputscn to send the home, clear screen cursor left, and erase to end of line, etc., to the screen. ChrPtr is a table (but not a jump table) of pointers to the various characters on the keyboard; this table is, of course utilized by bput et al. KyputB gets a character from the keyboard buffer and places it in the b register. Scroll moves the screen up one line, while FxCh Buf adjusts the character buffer pointer by one position. This buffer is in locations 0400 to 0450.

DATA REGISTERS and F000 BLOCK

e810	PIA1	to e813	e820	PIA2	to e823
e840	VIA	the 6522 chip to e84f	e880	CRTC	The CRT controller
eff0	ACIA1	the 6551 to eff3	eff4	ACIA2	the 6850 to eff5
eff8	SysLth	the System Latch	effc	BkSel	The Bank Select Latch

f03d	Nocndo*	"unable to perform req"	f0bf	EntMon	The Monitor entry point
f123	Mongret*	"Waterloo microMonitor"	f41d	MonErr*	Monitor error messages
f6c2	Ermsgs*	Error messages	f6e0	inttxt*	"Interrupt"
f6ee	AzWds1*	Assembler keywords	fcfe	AzWds2	more Assembler keywords
fdcc	Invld*	"Invalid command"	ff80	SysRst	System reset routine
ffa2	SysIRQ	Interrupt handler	ffb1	ExtIRQ	The RTI instruction
ffb2	SvRSV	Service 'RESERVED'	ffb4	SvSWI3	Service SWI3
ffb6	SvSWI2	Service SWI2	ffb8	SvFIRQ	Service FIRQ
ffba	SvIRQ	Service IRQ	ffbc	SvSWI	Service SWI
ffbe	SvNMI	Service NMI	fff0	RSVD	Vector 'RESERVED'
fff2	VSWI3	Vector SWI3	fff4	VSWI2	Vector SWI2
fff6	FIRQ	Vector FIRQ	fff8	IRQ	Vector IRQ
fffa	VSWI	Vector SWI	fffc	NMI	Vector NMI
fffe	RESET	Vector RESET			

Thus with a lot of help from many people, including much help from Waterloo for not treating SuperPET owners like a bunch of naughty kids and hiding the goodies from us, we now have an excellent start on the map of the 6809 side of the SuperPET. There is still plenty room for exploration; for the very ambitious, there are the still-unmapped languages. Who will find the equivalent of CHRGET in the 6809 side?

To all those who contributed: please send in your name for the routine or routines which you discovered. As more discoveries are made, we'll update the map. If you send an address, please check it; send the correct address, not the address at the start of your hex dump line, but the exact starting location. It is a chore indeed to verify, and I would deeply appreciate the help. Please send all updates and new locations to me, at the address above.

****

TELECOMMUNICATION for the SuperPET

by Jeff Larson, Route 1, Box 261D, Rustburg, Virginia 24588

Telecommunication in the micro-computer world is simply the process of getting computers to talk to each other. Aside from various computer networking schemes (which get complicated fast), the most common means of communicating is by a modem which converts the serial transmission of data from the RS-232 port to an audio signal which can be received by another modem through the telephone line. The proper disposition of this data depends on the communications software used by both computers. In other words, some computers expect certain responses during the transmission of data (handshaking), and use differing ASCII screen control codes to display the data. If the communications software is not suited for the computer on the other end of the phone line, the computers won't respond properly to each other and many strange characters along with unusual screen formatting will occur. The SuperPET is unique among microcomputers in that it has a telecommunications program built in (along with an RS-232 port). This makes it especially easy to get started.

I have been successful in talking with a Control Data mainframe computer and a Datapoint minicomputer using an assembly language modification to the built-in passthru mode routine so that I could save whatever was transmitted to me on disk. I had to use assembly language to operate at 1200 baud, since any non-compiled BASIC program is barely fast enough for 300 baud. The benefits: I can use the mEditor in the SPET to modify documents and programs and then ship the end result back to the host computer. The process has also made it possible to receive large amounts of data (6000 addresses) for input into a mailing list program. The same technique could be used to send programs and information back and forth between fellow SPET users.

A modem is the only thing that you need to start communicating. For right now, I will assume that the passthru mode (accessed from the monitor) is adequate to accomplish what you want, and I will concentrate on hardware. There are nearly as many modems on the market as there are micros, but for the most part, they all fall into one of a few distinct classes. Some are designed to be used with Apples, TRS-80's and other specific computers (not what you want). Others hook up to nearly any computer with an RS-232 interface (what you want). Some communicate at 300 baud only (slow - as in cassette speed). Others are capable of 1200 baud or both 1200 and 300 baud (1200 baud is 4 times faster, and is at the limit of Ma Bell's capability). Most use Bell 103 protocol for 300 baud, and Bell 212 protocol for 1200 baud (protocol being how the serial signal is converted to an audio signal). I have used others (like Racal-Vadic), but I would recommend sticking to the Bell 103 and/or 212 types for compatibility with the most equipment. Several popular brands of modems have been reviewed recent- in one of several micro magazines. I have used the Hayes Smartmoden with 1200 and 300 baud capability with great success; it has many nice automatic features (auto dial, repeat last dial, etc), and is about as cheap as a 1200 baud modem comes (discounted to $520.00). 300 baud modems are cheaper, mostly below $200, and many approach $100.00. For serious work, I'd strongly recommend 1200 baud--it's sort of like going from casette speed to disk drive speed, and at long-dis- telephone rates, that can make a difference fast!

Telecommunicating is not without problems, however. I found, for example, that SPET drops some characters at 1200 baud whenever the screen scrolls (it takes .035 seconds to scroll, which is enough time for about 4 characters to go by). In addition, some host computers need ASCII input from you, which you normally can't send; ESC and STOP (ASCII 03 and 27) are examples of keys which are processed by SPET instead of sent to the other computer. And it becomes highly desirable to save the info you receive on disk, and maybe to have the ability to send a disk file to someone else. Here is where special software packages are needed to add to the 'dumb terminal' capability of the passthru mode. You will encounter another problem, probably at 1200 baud and over very long distances: garbling of characters by the telephone line. Thunderstorms are especially good at contributing to the data you receive at your SPET. I haven't run into any problem like this even when talking to West Va. from Central Va. (some states don't have the best phone networks).

Here's a way to get started with Telecommunications using SuperPET:

1) Obtain a modem and become familiar with it. If you have a smart modem like the Hayes unit, you can verify correct operation by sending it commands in the passthru mode. The cable for the two modems I have used both work with the SuperPET without modification--direct connection of pins 2,3,4,5,6,7, and 8. If If you have difficulty getting the SuperPET to talk to the modem, make sure you are satisfying the handshaking connections for SuperPET.

2) Obtain instructions for 'logging in' from whoever you are to call. (See 'Commodore', March 1983, for info on Compuserve and telephone #'s, etc.).

3) Use the setup mode to set the baud rate to either 300 or 1200 baud--for whatever your modem is set for, and for whatever's expected at the other end.

4) Enter the monitor mode (from any language, or from the main menu).

5) Press p for passthru mode--from now on, any key you press should be sent to the other computer, and any transmission from the other computer should show

up on your screen. To verify whether you are sending or receiving characters, watch the LED indicators on your modem light up when it receives characters. If you get no response, check the RS-232 cable connections. (Modems usually require a different cable configuration than printers. See (1) above.) If you get double characters or other bad output on the screen, follow the modem manual to fix it.

6) Dial up the other computer (from the keyboard if autodial). Most 1200 baud modems will be direct connect (the modem connects directly to the telephone jack, bypassing the phone handset). Other modems may use an acoustic coupler, two rubber cups which fit over the telephone earpiece. The acoustic coupler type is usually dialed manually, whereas the direct-connect type often is dialed from your keyboard.

7) Follow instructions for using the other computer. You can get instructions for Compuserve, for example, along with access authorization, from a Radio Shack dealer. Ask for cat. no. 26-2224 and ignore any protests that you need software (the passthru mode takes care of your needs). If you operate at 1200 baud using the passthru mode, be sure to tell the other computer to add 5 nulls (ASCII 0) after each carriage return to avoid dropped characters. If you want to use the 6502 processor instead of the 6809, the programs to use 6502 mode are on SPUG disk 1 (the Commodore public domain programs). [Ed. We'll cover the 6502 packages in later issues.]

8) If you want the ability to send and receive files at 1200 baud using the 6809, send me a donation (enough to cover disk, postage, time & effort) [Ed. Ten $ U.S., Jeff?]. I'll send you an assembler-based program that has worked with a CDC mainframe and a Datapoint minicomputer. It is an extension of the passthru mode which opens disk files for you, and eliminates the need to use nulls while at 1200 baud. I have an 8050 drive, and cannot provide 4040 format.

[Note: This is the first of a series on TC. We hope to continue it with material on 6502 TC next issue, courtesy of John Frost of Seattle. For those interested: Associate Editor Terry Peterson published an excellent program called 'Smarterm' in the April issue of MICRO, which does what the name implies--makes a smart terminal out of SuperPET in 6502. In assembly language. Ed.]

nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn

(C) 1983 ◇◇◇ *THE APL EXCHANGE* ◇◇◇ *STEVE ZELLER*

uuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuu

Last time, we indicated that functions represent a very powerful tool in APL. Today, we will begin to examine their design and use in the language. There are six allowable forms of functions, summarized in Table 2.1.

Table 2.1 Form of Functions in APL

NAME	NUMBER OF ARGUMENTS	NO. OF FORMS	SYNTAX: NO EXPLICIT RESULT	SYNTAX: EXPLICIT RESULT
NILADIC	0	2	FN	RES <- FN
MONADIC	1	2	FN RHA	RES <- FN RHA
DYADIC	2	2	LHA FN RHA	RES <- LHA FN RHA

When you declare a new function to APL with the 'del' operator, you also specify the form the function will take, along with its arguments and its local variables, if any. This is called the function's header; it is line [0], and you can edit it when you re-open the function (you must do so for explicit results in some functions which follow). We must start by thinking carefully about the form the function will take. In particular, we need to consider how the function will communicate with other functions in the workspace and with the APL language primitives. We'll consider some examples of this shortly.

A function (FN) may or may not return an explicit result (RES) and it can have no arguments, a right-hand argument (RHA) or a left- and right-hand argument (LHA and RHA). Think of the arguments as "dummies" that only take on values when the function is invoked. They have significance only to the function while it is executing and are not available after execution is complete. (Note: this is different from arguments in a FORTRAN subroutine call.) Information from the function can be returned as an explicit result, however, by including still another dummy argument in the header (RES) along with the "gets" symbol. Variables used during the function's execution but which are to be "hidden" from the workspace are called local variables and are also declared in the function header.

Most of the variables used in a function should be declared as local variables. This will keep the workspace clean and avoid name conflicts as other functions are added. All other variable names appearing in the function are global variables and can be used to pass information back and forth between the function and the workspace. Think of global variables as key variables to your system of functions in the workspace. The width of your printer, for example, would be a good choice for a global variable.

Keep the function short. A modular approach, with each function performing just one simple task, will allow you to use functions in several contexts. Breaking the functions down into short modules makes debugging simpler and editing easier. In general, have the single task performed by the function returned as an explicit result. This provides much more flexibility in combining functions to accomplish more complicated tasks and in directing output to the printer. In example 2.1, we see four functions that have no arguments. The two functions which do not return an explicit result, HELLO and GOODBYE, cannot be utilized in any indirect way, while the two functions that do return explicit results, FIRST and LAST, can be combined to yield a composite result.

```
EXAMPLE 2.1: NILADIC FUNCTIONS                                        ACTION
   A...∇HELLO                    | B...∇GOODBYE                   | TYPE:
    [ 0]    HELLO                |  [ 0]    GOODBYE               |   HELLO
    [ 1]   'HELLO YOURSELF '     |  [ 1]   'NICE TALKING TO YOU ' |   GOODBYE
                                 |                                |
   C...∇FIRST                    | D...∇LAST                      |  TYPE:
    [ 0]    R ← FIRST            |  [ 0]    R ← LAST              |    FIRST
    [ 1]   R←'STEVE'             |  [ 1]   R←'ZELLER'             |    LAST
                                 |                                |FIRST,' ',LAST
```

Now let's consider functions with one or two arguments. Monadic functions are shown in example 2.2. Note that returning each function's result explicitly allows them to interact naturally with each other and with APL.

```
EXAMPLE 2.2: MONADIC FUNCTIONS |      ACTION
   A...∇SUM                    |  TYPE:
     [ 0]    R ← SUM X         |      SUM 1 2 3 4 5 6 7 8 9 10
     [ 1]   R←+/X              |      SUM ⍳10
                               |      SUM 10,SUM ⍳10
```

```
  B...∇LOG                      |
    [ 0]    R ← LOG X           |   LOG 10
    [ 1]   R←⍟X                 |     LOG ⍳10
                                |   LOG SUM ⍳10
  C...∇EXP                      |
    [ 0]    R ← EXP X           |   EXP 10
    [ 1]   R←*X                 |   EXP LOG 10
                                |   LOG EXP 10
```

Functions with two arguments, the dyadic form, appear in example 2.3. They are of two basic varieties. In one instance, each argument is of equal "weight" in the function. In the other, the left-hand argument is used to modify the function's operation on the right hand argument. This latter approach to the dyadic function's form is not strictly enforced in APL, of course, but is good programming practice.

```
EXAMPLE 2.3: DYADIC FUNCTIONS     |          ACTION
   A...∇PLUS                      | TYPE:
     [ 0]    R ← X PLUS Y         |    (⍳10) PLUS ⍳10
     [ 1]   R←X+Y                 |    SUM (⍳10) PLUS ⍳10
                                  |    LOG 5 PLUS ⍳5
   B...∇UNION                     |
     [ 0]    R ← X UNION Y        |    100 UNION ⍳5
     [ 1]   R←X,Y                 |    FIRST UNION ' ',LAST
                                  |
   C...∇INTERSECT                 |
     [ 0]    R ← X INTERSECT Y    |    FIRST INTERSECT LAST
     [ 1]   R←(X∊Y)/X             |    5 INTERSECT ⍳10
                                  |
```

The functions in these examples are trivial. However, the ideas behind their design and use apply to much more complicated functions. Consider HIST, shown in example 2.4. It produces a histogram from a vector of frequencies. The function does just one thing: returns a character matrix of the histogram. This allows the result to be sent to a printer or used in some other way.

```
    ∇HIST∇
[ 0 ]     M ← HIST F ;NMAX;MAX;NF;NPLOT;LINE
[ 1 ]    ⍝GENERATES FREQUENCY HISTOGRAM: DUE TO SMILLIE (P. 20)
[ 2 ]    NMAX←MAX←⌈/F←⌊0.5+F
[ 3 ]    NF←⍴F+⍴M←⍳0                        ⍝ TO EXECUTE, TYPE:  HIST ⍳10
[ 4 ]    S1:NPLOT←(MAX≤F)/⍳NF               ⍝ NOTE:  SCALING MUST BE DONE EXTERNALLY
[ 5 ]       LINE←NF⍴' '                     ⍝ TRY:' HIST (⍳20)÷2
[ 6 ]       LINE[NPLOT]←(⍴NPLOT)⍴'⎕'
[ 7 ]       M←M,'.',LINE
[ 8 ]    →(0<MAX←MAX-1)/S1
[ 9 ]    M←M,(1+NF)⍴'.'
[10 ]    M←(1+NMAX,NF)⍴M
```

In example 2.5, we compute summary statistics for a vector of data. This task is broken into two functions. DOSTAT does the needed calculations and SHOWSTAT displays the results and takes care of the formatting. This allows DOSTAT to be used with other routines that may require one or more of the summary measures.

```
   ∇DOSTAT∇
[0 ]     STATX ← DOSTAT X ;R;MAX;MIN;N;MEAN;VAR;SD;MD;MED;MODE;V;M
[1 ]     ⍝COMPUTES SUMMARY STATISTICS FOR VECTOR X
```

```
[  2]   ⍝DUE TO SMILLIE [1969], P. 16
[  3]   R←(MAX←X[ρX])-MIN←(X←X[⍋X])[1]
[  4]   SD←(VAR←(+/(X-MEAN←(+/X)÷N)*2)+(N←ρX)-1)*0.5
[  5]   MD←(+/|X-MEAN)÷N
[  6]   MED←0.5×+/X[(⌈N÷2),1+⌊N÷2]
[  7]   →(N>ρMODE←((ρV)ρ(ιM)≤1)/V←X[(V=M←⌈/V←+/X∘.=X)/ιρX])/S1
[  8]     MODE←ι0
[  9]   S1:STATX←N,MAX,MIN,R,MEAN,VAR,SD,MD,MED,MODE

∇SHOWSTAT∇
[  0]    R ← SHOWSTAT STATX
[  1]   ⍝FORMATS OUTPUT FROM: 'DOSTAT'
[  2]   M1←3 14ρ'SAMPLE SIZE   MAXIMUM       MINIMUM       '
[  3]   M1←M1,[1]3 14ρ'RANGE         MEAN          VARIANCE      '
[  4]   M1←M1,[1]3 14ρ'STD. DEVIATIONMEAN DEVIATIONMEDIAN        '
[  5]   R←M1,10 4⍕9 1ρSTATX
[  6]   →(9≥ρSTATX)/0
[  7]   R←R,[1]1 24ρ'MODE          ',10 4⍕STATX[10]

TYPE: DOSTATι10    OR    SHOWSTAT DOSTAT ι10
```

An excellent APL disk from Australia recently came to SPUG via Waterloo. It consists of tools for use in Exploratory Data Analysis (EDA) and contains eleven basic workspaces, extensive documentation, and examples. Almost all of the 8050 floppy is filled with this material. The techniques were developed by John Tukey of Princeton, who in his textbook of the same name (Addison Wesley, 1977) characterizes EDA as 'numerical detective work.' The disk supplements 'Interactive Data Analysis: A Practical Primer,' by D. R. McNeil (John Wiley & Sons, 1977). McNeil and a colleage, M.P. McFarlene, authored the disk. I found their book in paperback, but it was very expensive ($22). If you want a copy of the disk, send $10.00 U.S. to me (no disks, please) for the 8050 version. If you want 4040 format, send $20.00 to the Secretary, Paul V. Skipski, for the three-disk set. This material, being a gift and in the public domain, is offered to all--not just to members of SPUG. Make checks out to SPUG.

Reference: Smillie, K.W., 'Statpack2: An APL Statistical Package,' Dept. of Computing Science, University of Alberta, 1969.

∩∩∩

6425 31*ST ST., N.W., WASHINGTON, D.C.* 20015 *U.S.A.*

∪∪∪

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

No. 1: not available	No. 4: $1.25	No. 7: $2.50
No. 2: $1.25	No. 5: $1.25	Send check to the editor at
No. 3: $1.25	No. 6: $3.75	PO Box 411, Hatteras, NC 27943

Add 30% to prices above to cover postage if outside North America.

****

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

APPLICATION FOR MEMBERSHIP, SUPERPET USER'S GROUP

Name:______________________________ Disk Drive: _________ Printer:___________

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

For Canada and the U.S.: Enclose Annual Dues of $15:00 (U.S.) by check or money order, payable to SPUG. DUES ELSEWHERE: $25.00 U.S. Mail to: Paul V. Skipski, Secretary, SPUG, 4782 Boston Post Road, Pelham, N.Y. 10803, USA.

Newsletter published by the SuperPET Users' Group (SPUG): editorial offices at PO Box 411, Hatteras, N.C. 27943. Secretary, Paul V. Skipski, 4782 Boston Post Road, Pelham, N.Y. 10803. Membership applications, dues, and inquiries to Mr. Skipski; newsletter material to Hatteras, attn: Dick Barnes, Editor. SuperPET is a trademaark of Commodore Business Machines, Inc.; WordPro a trademark of Professional Software, Inc. Contents of this issue copyrighted by SPUG, 1983, except as otherwise shown; reprinting by permission only; SPUG members are authorized to use the material. Enclose a self-addressed, postpaid envelope with all material submitted and all inquiries requiring reply. Membership: $15.00 per yr. U.S. in North America, $25.00 overseas and elsewhere. See enclosed application.

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. Canadian members: send Canadian dimes or quarters for postage, but no paper currency.

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

HATTERAS, N.C.
JUN
2
PM
1983
27943

PRINTED MATTER