The latest computer shows indeed revealed a few new computers and a least 1500 more printers; we will soon have one model of printer for every user in the U.S. and Canada--each, of course, with its own peculiar command set. We hope the printer makers either adopt a standard command set or that 99% of them go bankrupt, if only to save the sanity of the poor devils who must write a full set of printer drivers for all commercial software. Users, of course, are in the position of the chap who, having saved the life of the Sultan, was rewarded with one night in the harem amongst a thousand beautiful houris. He went mad from indecision.

Atari did reveal its MC68000-based ST-series, with a Mac-like GEM interface between the operating system and the user; 512K for $599 has a nice ring to it. Commodore showed the C-128, which contains three different microprocessors, one to run CP/M, one to match the C-64 in all operating details, and one to handle an 80-column screen plus the full 128K of memory. It essentially is a 6502/Z80 machine. Rumors abound about Commodore's 68000 machine (the Amiga), but nothing much seems known.

We hear (happily) that IBM is reported to have abandoned the notion of 3.5-inch drives for the new PC II, after a survey showed users didn't like the idea. We have about as much use for cutesy wee disks as we have for measles, never having been able to cram enough files on a 5-inch disk. And we don't like the price of the wee ones--three to four times as much as a 5-inch disk, with less capacity. Having lost only one 5-inch disk in three years, with moderate care and from six to eight hours of computer operation per day, we scoff at the hype of "more protection against dirt." It's the same old "new and improved" jazz the hucksters employ to sell soap. The cutesy drives may be fine for portables, but we want a 5-inch drive with 1.3 to 2 megs on it--sized for mankind, not elves.

The e: date on your address label means ENTRY, not expiration...

We got an angry note from young Bodsworth: "Why didn't you tell me my membership had expired!" His e:date is 9-1-84; his membership won't expire until 9-1-85. We always tell you when time is up with a note on your address label, underlined in red, saying: "Membership EXPIRED!". We love to see renewals come in sooner.

If thy label is redmarked this issue, 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 the usual fees. Send an article while you're at it.

FIRST THE SCALPEL, THEN THE OPERATION And to make a scalpel, you first find some iron ore, charcoal, and limestone, and then you build a furnace... Guru John Toebes started to reassemble microBASIC, using the present assembler and linker, only to find himself without sufficient memory and in hours-long sessions with the huge files. So he had to back off and rewrite both assembler and linker. Progress to date: most files assemble in one half to one-quarter of the time previously taken; his FORTH package formerly assembled in four hours; it now assembles (with lots of macros) in fifteen minutes. And that package is a small one, compared to about 36K bytes of mBASIC code. Once assembler and linker are working as they oughta, he must reassemble mBASIC in packages and then link the same way, for there's no way to finish a 36K byte program in 32K of user memory. And there's no way to start writing a compiler until mBASIC is reassembled and working properly. First, the scalpel...

ISPUG UTILITY DISK II
Fewer Programs--More Instructions

After a prolonged pregnancy, ISPUG finally gave birth to Utility Disk II, holding some brilliant offspring, thoroughly documented--most with tutorials as well as instructions. The prize baby of the lot is BEDIT, of which we've said much, and on which you'll find separate articles this issue. We show below a partial directory of the programs; note the size of the tutorial and instruction packages. We're convinced much good software languishes unused because the instructions are lousy; we tried to overcome that problem; shown below is a partial directory of the new disk:

20	"dos:men"	PRG	See comments below
19	"dos_instr:e"	SEQ	
24	"copy/kill:men"	PRG	See comments below
22	"copy/kill:ein"	SEQ	Instructions for COPY/KILL (CK)
5	"clip:men"	PRG	Converts 6809 ASCII files to PaperClip format
8	"clip_instr:e"	SEQ	Instructions for "clip"
49	"bedit1.0	PRG	Joe Bostic's new Editor
116	"bedit_instr:e"	SEQ	The instructions for BEDIT
96	"bedit_tut:e"	SEQ	A tutorial on the new features of BEDIT
79	"batch_tut:e"	SEQ	A tutorial on BATCH files in BEDIT
64	"bedcalc1.0"	PRG	BEDIT with CALC built-in
16	"calc"	PRG	Interrupt version of CALC; use with other Editors
12	"calcs"	PRG	Short form for use in all languages/facilities
69	"calc_instr:e"	SEQ	Instructions for CALC (all versions work the same)
94	"calc_tut:e"	SEQ	A tutorial on all versions of CALC

To save space, we list above about half the programs. The material almost fills two 4040 disks.

Two of the programs above deserve mention here: First is Alain Proulx's DOS, a machine-language program which runs at main menu. It handles all DOS functions; lets you read any SEQ file on the disk examined, and is menu-driven. For those who find DOS work difficult, this is a DOS manager which lets you perform any function you want, from COPY to FORMAT, from RENAME to SCRATCH, without having to memorize or type in the DOS commands. Important note: Alain has produced two versions, one in French and one in English. His instructions likewise are in both languages. Please state which version you want: French or English! If you do not specify, we'll send the English version.

Second is Loch Rose's COPY/KILL (CK), a machine language program which loads and runs at main menu. As with DOS, you may read any SEQ file on disk. You may mark any file to be copied or scratched with the touch of a key; even better, you may move the filenames with other keys to define the order in which the files are to be copied. CK is menu-driven, simple to use, and fast. You may change your mind as to which files are to be copied or scratched and switch the order in which the files will be copied with a keypress or two.

Between DOS, CK, and the BATCH files capability in BEDIT which we define in a separate article this issue, anybody who switches to 6502 for DOS work is not only out of his or her ever-lovin' mind but driving a Model T whilst a Porsche lingers unused. We are now spoiled rotten, and haven't typed a list of filenames

for DOS work in the past six months... Final note: you needn't know machine language from a mackerel to use any of the programs.

NOTE: As we print this issue, BEDIT does not support "puts" to HOST; Joe Bostic is hard at work upon the problem. Most of us do not use the HOST capability; if you need it, wait for a later version.

Please read the article on BEDIT's capabilities in this issue. We suspect that the fully selective directories (di *.asm lists all .asm files on disk) and the sorted directories (cat *.asm lists all .asm files in alphabetical order) will alone be worth the cost of the disk, which, as usual for documented ISPUG disks, costs $10 U.S. in 8050 format, or $16 for two 4040 disks. Order from ISPUG, at PO Box 411, Hatteras, N.C. 27943. This is the best disk we've yet issued.

ONCE OVER LIGHTLY
Miscellany

Guru Terry Peterson reports on a way to use your printer off the serial port and off the IEEE bus without touching the "linefeed with CR" switch on the printer. As most of us know, Commodore machines don't send a linefeed with a CR; your printer must generate the linefeed. If you adjust the printer to produce a linefeed off the IEEE bus, and then try it on the serial port, you'll get two linefeeds, one from the "p serial" routine, and one from the printer.

Terry's tip: You can kill the extra linefeed sent from the serial port by poking byte $7E in the zero page to a 1 (it normally says 2). This location defines the number of bytes the serial port should transmit at line-end. If you do this, only the first byte ($0D) is transmitted; the second ($0A, or linefeed) is not.

SETTING MEMEND_ IN THE ASSEMBLER John Toebes sent us a report on a bug in the Waterloo Assembler. If you're using an ML module with it, loaded up high in user memory, be sure to set MemEnd_ for that module at least one byte lower than the start of your program. It seems the Assembler code reads the set MemEnd_ and, depending on whether the address is odd or even, may adjust itself to use memory one byte past MemEnd_. Be safe. Allow for it.

NOTE ON WORDCRAFT ULTRA Frank Avenoso of 288 River Road, St. James, New York 11780 writes that he is much pleased with the performance of Wordcraft Ultra for the CBM 8032/8096 (it costs $400). At that price, Frank says, it should and does offer a lot: "You get the usual WordPro/PaperClip features without global search and replace but plus true proportional spacing with left and right justification and bold print, etc. It makes for nice in-house manuals with the type-set look. I could ask for only one additional feature--to take advantage of the extra memory in SuperPET. Any readers who want more informtion should drop me a line." Cimmaron Corporation of California distributed the program until a few months ago; now you must purchase from Dataview-Wordcraft, Colchester, U.K.

ON SUFFOCATING VERBS The computer press continues to disgorge undigested mush written by programmers who are proud of their tight computer code but can't write English. We discovered why--it's the current fad to suffocate all living verbs:

We illustrate with a recent massacre less bloody than most, and underline the slain verbs: "Any register capable of storing retrievable data is usable for passing data to and from an assembly-language routine." Four verbs lie dead, suffocated with "ble" and "ing" in the current fashion. What does the mush say?

"In assembly language, pass data in a register you can write to and read."

How else would you pass it, in heaven's name? A platitude lurks there, hidden by the bodies of verbs smothered to death with pillows of "ble" and "ing".

LINKING WITHOUT CRASHING We tried to link a program which took up four banks (about 16K bytes) and crashed, memory full, when we were halfway through the third. The command file referenced not only watlib.exp but a long usrlib.exp file of over 90 blocks. When we cut that file down to only the exports we needed in the program, it occupied only 9 blocks--and we were able to link a longer program (about 18K bytes or 4.5 banks) with no further problems.

It turns out that ".exp" files occupy a large amount of memory. If you run out of memory whilst linking, edit watlib.exp and usrlib.exp files down to only the system routines or locations you must use. You can save even more memory by putting such routines/locations into equates in your .asm files. The linker is then not required to keep a table cross-referencing locations and labels. As Terry Peterson notes, you can also save considerable memory by using short names for labels which are XREF'd (and put in tables internally). Cut a label name from "firstbuffer" to "buf1" and you save at least seven bytes...

There's another way to approach the problem: put your exports in your command file, in the form: "export wunderbar = $0000". This won't save as much memory as putting equates into your .asm files, but it may be more convenient than making up a special "usrlib.exp" file.

KEYSENSING BUG Please hold down both SHIFT keys on SuperPET, and with your third hand, press these keys: _, !, ", TAB, ESC, A, OFF, and Z. Hmmm. The keys listed either do not print at all or print the wrong character. Now, depress SHIFT LOCK and hold the right-hand SHIFT key down. The same thing happens. What's wrong? The keyboard-sensing routine in ROM can cope with a shifted keyboard only when one SHIFT key is down, but not with both down. The SHIFT LOCK key acts only to enforce the left SHIFT key, not the right--and the two SHIFT keys return different codes at the PIA, which senses keypresses.

SuperPET does not know what to do with the leftmost keys on the board when SHIFT LOCK and right SHIFT are down, or when both SHIFT keys are depressed. No, there is nothing wrong with your particular machine. The problem afflicts us all. If you're curious, transfer to 6502 and do the same thing; try SHIFT 2...

USED SUPERPET AVAILABLE? John E. Stump of PO Box 1112, Wiley Hall, West Lafayette, Indiana 47906, has a disk drive and a B-128 but wants wants wants a SuperPET. If you'd care to sell one, write him!

LOADING BIG CHUNKS of PIC In language, that is. Guru Terry Peterson reports no luck trying to load programs of more than 512 bytes into Fortran or Pascal (See Vol. II, No. 2): "mFortran seems to suffer from the bug V1.1 mBasic used to have; it defaults to a maximum record size of 80 when no size is given. Worse yet, it cannot be circumvented by POKEing, the way my mBASIC patch program did (II, 29), because the file doesn't actually get opened until the first READ. Time for another mFortran patch? The situation in Pascal is even bleaker; its READ procedure won't accept string variables as arguments, so you have to fetch the bytes one...at...a...time. Slow. When I tried it, I got only 130 bytes of the 1000 in the file. Dunno why."

OS-9 DISTRIBUTED Just as we were finishing copy for this issue, UPS delivered a fat package of manuals, instruction sheets, and a small adapter board for our 2-board SPET. Super OS-9 for SuperPET, version 0.9, has been shipped. We've not had time to solder in the kit, but we did manage to get halfway through the manuals; we have concluded that OS-9 is the poor man's version of UNIX. TPUG promises a screen editor called SCRED soon; we're pleased, for the editor which comes with OS-9 is a line editor only, and (if you follow the horses) seems to be out of Terminal by Teletype. Bill Dutfield of TPUG reports that BASIC 09 is really a superset of mBASIC and very efficient; it ran a test program in one-third the time of BASIC 4.0 and one-fifth the time of mBASIC. Well, we'd expect the 6809 to do a good, fast job on a language tailored for the chip--which none of the SuperPET languages are, all having been ported to the 6809 from a high-level language (WSL, we suspect) and not optimized for the 6809. Take note of those speed differences, Hal; it confirms what you've been preaching.

We hear that 83 OS-9 kits have been shipped to those who ordered early, in the hope that these early users will find and report whatever bugs may exist before version 1.0 is issued. We have a report from Russ McMillan of Madison, WI that a computer dealer installed his OS9 board and that SuperPET failed to work; when the OS9 gear was removed, Russ's SPET still wouldn't work. Board damage? We dunno. On the other hand, Associate Editor Stan Brockman installed the OS-9 gear himself, and it's up and running. We hate to carp at TPUG after the tremendous effort to get OS-9 on SPET, but the first diagram and instructions on how to install the kit on two-board models aren't clear or easy to follow.

We reported both problems to Gerry Gold of TPUG, who said he'd have Russ McMillan's board fixed at no charge if Russ will ship it; he reports a photo is being made to replace the unclear diagram in the instructions. Obviously, TPUG is supporting OS-9. If you have problems, call or write TPUG, 1912A Avenue Road, Suite 1, Toronto, Ontario, Canada M5M 4A1, (416) 782-8900. OS-9 is available from the same address for $195 (Canadian) to TPUG members or U.S. Associates. Note: the OS-9 kit for three-board SuperPETs requires no soldering; it plugs in.

CHECK THAT ESCAPE SEQUENCE Associate Editor John Frost had a simple problem which almost cost him $600. He was printing hard copy from WordPro when his DIABLO simply quit printing visible characters--though it continued to make all the suitable noises. Hmmm. The ribbon was positioned too low for the daisy wheel to strike it. The local service center said that a replacement printhead mechanism would cost $600. John balked and instead taped the ribbon cartridge in the proper position. A friend suggested that John try the ESC sequence which resets the ribbon to <u>not</u> print red (DIABLO lets you shift the printhead so the wheel types a second color from the top of a wide ribbon). John did; farewell tape; hello $600. Gee, howcum the service center never mentioned a software problem?

HOPE SPRINGS ETERNAL Commodore owners still write letters to Commodore and expect an answer. Lessee, now. We reckon there are 5.5 million owners of VICs and C64s, and a minimum half a million more who own the other Commodore models; say six million total. Suppose each owner writes one letter per year to Commodore and that a good letter-answerer can reply at the rate of 48 answers per day (one reply every ten minutes). Commodore would need a staff of 569 letter-answerers. If salary and overhead were a modest $30,000 each, letter-answering would cost over $17 million per year. If Commodore nets $20 for every C64 sold, it would have to sell 850,000 of 'em to recover the cost of writing letters--and then answer 850,000 more letters... A Ponzi scheme in reverse, no less.

LONG FILES, SHORT EDITOR Readers keep asking how in the world to edit files too long to fit in the mED. A typographer asked what to do; a teacher couldn't load more than the first section of huge files of processed census data. The solution is simple, once you think about it--copy a fixed packet of lines to a new disk file, then close it and open a second, then a third, etc. If the new filename is "newfile", you get a series of numbered files on disk, such as the list below:

```
open #30,"oldfile", input    ! Read the long file
open #40,"newfile", output   ! And make a new one
on eof ignore
loop
  linput #30, line$          ! Get a line
  if io_status then quit
  print #40, line$           ! Print it to new file
  linecount=linecount+1      ! Count the lines
  if linecount=300           ! End the 300-line packet
    packet=packet+1 : close #40
    open #40, "newfile" + "." + value$(packet), output
    linecount=0
  endif
endloop
```

```
newfile
newfile.1
newfile.2
...
newfile.n
```

And, of course, you can get any single one of the new files into the mED to edit. If you should want the files back in one big one, for whatever reason, it is simple to write an "append" program which joins them again, or to concatenate them with DOS commands. We've dissected a number of biggies this way, and have put 'em back together again without scars.

THE COMPLEAT EDITOR We've announced in recent issues a new Editor for SuperPET which uses the same commands as the old mED but adds many powerful new ones, is free of the bugs which inhabit Waterloo's, and is fast. Find a summary below of the major improvements in BEDIT (the "B" is for the last name of the author, Joe Bostic of Las Vegas, to whom we all are deeply indebted for this jewel).

1. SELECTIVE Directories: Want to see the titles of just the .asm files on your disk? Try: di *.asm. Yes, the asterisk prefix is legal. How about all the files named "whatsit.xxx"? Try: di whatsit.* . Or, if you want a list of files with "sit" in them, use: di *sit*. Yup. Also legal. You may, of course, employ the ???? wild cards also.

2. STOP does! Ever send seven pages or so to printer, and then accidentally kick off the print command for the second time? In old mED, you either turned SPET off or (sob) reprinted seven pages.... We love Joe for arranging BEDIT so the STOP key STOPs all "puts" and "gets" and all searches or changes.

3. STOP Scrolling: So you have a file in BEDIT, and want to read one which is on disk. The TYPE command not only prints the file to screen but stops or starts its scrolling at a touch of the spacebar.

4. IMMORTAL Filenames: Neither disk number nor file title change when you "put" or "get" a file, whatever the destination, unless you yourself change that name or you clear the editor with "*d" and then get a new file. "Puts" to printer, in particular, do not throw away the name of the file in SPET memory.

5. IMMORTAL Drives: You control the default drive with the DEFAULT command. No matter where you get or put files, the default device is not changed--until you change it. Yes, the default device may be either drive on device 8, 9, 10...

With this arrangment, you never retype the title of your working file. If the default drive is 0, you simply "p" it; to drive 1 you "p disk/1".

6. SORTED or Unsorted Directories: The DI command gives you an unsorted list of files; CAtalog sortes them alphabetically--in both cases in two columns to the screen. DI sends single-column lists to printer and disk; CA sends sorted lists there in two columns.

7. DIRECTORIES stay on screen while you read them and write DOS commands!

8. MOVE Text, ECHO Text: Any text may be duplicated (ECHO) or moved elsewhere (MOVE). With MOVE, the original text is deleted; with ECHO, it's simply copied. What a time-saver!

9. WORD-WRAP: When you're in INSERT mode, any word which won't fit at the end of a line is wrapped to the next (except for words of more than 40 characters--such as a continuous line of dashes "------").

10. COPY is **Fast**: Wherever possible, Joe used the built-in 3.0 DOS commands, so that the turtle-like pace of mED's COPY command is no longer a burden. You also no longer must remember to add PRG, USR, or REL to filenames when you COPY files; they're copied in the format shown on directory.

11. FAREWELL "g ieee8-15": We now have @ (for Disk Command), so that any 3.0 DOS command may be entered directly at command cursor, as in: @ d1=0. For device 9, you say: @9 c1=0 or whatever. Yes, lower case commands are welcome.

12. CONTROLs in BEDIT Files: Old mED can't print any ASCII Control Code. In BEDIT, you may embed in files any control character, either in hex, its ASCII code letters (esc for ASCII 27, for example), or in decimal. If you want a carriage return in a file, you enter either <cr>, <$0d> or <13>. When you send the files to disk, you PRINT them so SPET knows what's to be done with the values. Yes, you can control any printer or plotter directly from BEDIT.

13. EXEC files: If you put an EXE file to disk, and then EXECUTE it, it will execute any command you can issue from the Editor just as though you typed it in yourself from the keyboard. See the article on BATCH files, this issue.

14. THE NOT Search: Gee, big list of files on screen and you can't find the .asm files you've been working on. Use *\.asm\d (which deletes all files from the list which do NOT have ".asm" in them). Well, there's your list...

15. UNIVERSAL Search/Replace. The %~ metacharacter in BEDIT lets you replace every occurrence of a phrase, whether at start of line, end of line, or embedded in code. With the %~ meta, you can change "place" to "prizes" without affecting "placeit", "emplace", or "place$".

16. MORE Goodies: These include "f" for "free"; give it at command cursor, and you read free memory. You can change text from upper to lower case or vice versa with the LOWER and UPPER commands. Last, if you ever wiped out a precious line of text with PF2, and wept because you could never get it back, be of good cheer: press SHIFT/RUN, and, as with Lazarus, your line is restored. And more.

WOOPS AND WHOA! A DOSBUG! Whilst we were smugly finishing an article on how to use BATCH files, we got a note from Joe Bostic about a bad bug in the 3.0 DOS command "c1:*=0:filename". Joe reported that if "filename" exists on the copy-to drive, the command above will crash SPET. We tried it. It did. We tried it again in Waterloo's mED, and crashed; in Toebes V1.3 mED, and crashed--and in 6502, and crashed. The bug in the 3.0 DOS is virulent! You always crash if a file with the same name exists on the drive you copy to. The DOS never gives a FILE EXISTS error message.

Joe reported the problem does not exist if the copy command is given in the format at the left, where a name replaces the asterisk on the left side of the "=" sign. So, if you are going to COPY, either copy to a NEWed disk, scratch files with the same name on the copy-to drive, or repeat the filename, as above. The last solution won't avoid a FILE EXISTS error message, but it will prevent a crash.

```
C1:name=0:name
```

HOW TO TAKE THE TEDIUM OUT OF TYPING or, BATCH FILES ILLUSTRATED Sooner or later, all of us confront the fact that our disk files are a mess. We know we ought to reorganize them, but flinch when we think of the time we'll spend copying files, one by one. Let all the lazy amongst us now rejoice. We have a swift, simple way to let SuperPET do all the monotonous work. Let's see how we do it in BEDIT.

First, we get a list of files to be copied by editing a directory put to disk with: di disk index <RETURN>, which forms a disk file named "index" on drive 0. In BEDIT, we may move any entry up or down a list with the MOVE command, so that we copy files in the sequence wanted. After editing, we store the list on disk as file "list:bed" (the suffix :bed identifies a BEDIT batch file).

Next, we must understand two new commands in BEDIT. EXECUTE executes every line in a disk file as if you entered that line from the keyboard. A second new command, "@", shows that you want to use 3.0 DOS commands, as in: @ s1:filename, which'll scratch "filename" on drive 1. The "@" replaces "g ieee8-15."

We now create on disk a permanent command file which we'll use forever after for all batch copying jobs. This file never changes; we saved ours to our language disk. We show the command file below; remember that it executes as if you typed in each line yourself. You may EXECUTE this file to copy any "list:bed" on disk simply by saying: ex scrcopy:exe <RETURN>. And it's fast.

Note we avoid the DOSbug reported elsewhere this issue by scratching all files of the same name on the copy-to disk as our first step. We named the routine below SCRCOPY:EXE for obvious reasons. It always copies from drive 0 to drive 1.

```
*d                          Step 1 is to get our list of files and convert it into
g list:bed                  a SCRATCH list for all files of the same name on drive
$                           1. This works even if the files don't exist on drive 1.
*c/%.%*"/@ s1:/
*c/"%.%*/                   The list "scrat:bed" remains in mED memory.
p scrat:bed
ex scrat:bed                We execute the scratch.
$                           Step 2, starting here, is to convert the SCRATCH com-
*c/s1:/c1:*=0:/             mands to COPY commands. The list is then filed as a
```

```
p copy:bed                COPY list.
*d
ex copy:bed               All files are automatically copied; then all working
scrat scrat:bed           files are scratched.
scrat copy:bed
```

Is what goes on a bit obscure, up in the CHANGE section of the file? Let's take a look at what happens when we: *c/%.%*"/@ s1:/, using the example at the left. %. stands for any character; %* for zero or more repetitions. We replace everything from the start of the line through the first quote. The second line at left shows the result; the DOS command now prefixes the line. In the next CHANGE line, we get rid of the last quote, the spaces, and SEQ. We now have the third line at left, above. After we've scratched files on the copy-to disk, we change the scratch command to a copy command--shown as the last line at left, above. Don't wince at the change process; you only type it in once.

```
27   "sum.asm"          SEQ
@ s1:sum.asm"           SEQ
@ s1:sum.asm
@ c1:*=0:sum.asm
```

SCRCOPY:EXE works with any list in which the filename to be copied lies between a pair of quotation marks--and ignores comments, filetypes, or anything else outside those quotation marks. But--there may be no other quotes on the line. Any of the entries below will be copied:

```
  12  "any_filename"    PRG       These comments will be deleted.
                "is_indented"     Copies okay.
Will copy? "a_file"               Yes, if on disk is a file named: a_file
```

In sum, copy the permament file SCRCOPY:EXE to your language disk. After that is done, you may forever after copy any list of files saved to disk as: "list:bed" by 1) copying SCRCOPY:EXE to drive 0, and 2) saying: ex scrcopy:exe <RETURN>. That's all there is to it--except for finding the dang fish if you go fishing whilst SuperPET does all the work. Quite obviously, you may create any number of EXE files to perform other jobs, even to copying full disks between devices.

Last, we received a most clever routine from Joe Bostic which compares the directory of the copy-to drive with the original list of files and prints to the screen a list of the files which were <u>not</u> copied to drive 1. If your files are split between separate disks, you can file the "missing" list and then copy the files from another source disk. CONFIRM:EXE is too long to print here, but is in the batch file tutorial found on ISPUG Utility Disk II.

CALC AND BEDCALC
or
A Calculator in That Kilobuck Computer

It always distressed us that we never could find our ASCII table when we wanted the code for "Z", that our calculator battery was dead when we had to convert decimal to hex, and that we always copied a long series of numbers from the screen, when summing, with a minimum of one error--after which, not having a tape, we had to re-copy the whole series again. Somewhere in the piles of papers around the computer is a table showing the graphics characters and their codes; somewhere a list of the control characters from @ to _, somewhere the table which converts hex to binary...somewhere lost, that is. In final frustration, we sat down and wrote CALC, which does all these things in SuperPET itself, which is a lot harder to misplace.

We made three versions of CALC so it can be used anywhere. We write this article in BEDCALC, Joe Bostic's editor with CALC in bank 10. The Editor doesn't know it is there until you need it. The results automatically enter Editor text.

CALC will: 1) Sum columns of digits, 2) Convert binary, decimal, or hex into any other notation, 3) convert ASCII code to a character (including our graphics set), or a character to ASCII code, 4) do integer arithmetic on hex, decimal, or binary alone or intermixed, and 4) do all floating point arithmetic. All you do is type your question and press PF4.

You must know, of course, that $ mean hex notation, D means decimal, % means binary, # stands for "number," and that @ stands for ASCII. If this doesn't blow your mind, the rest is easy. You enter numbers in the left-to-right order we all learned in the fourth grade. First, let's convert:

```
Questions:   D45671=$          $3210=%                    %0110 1100=$

Answers:     D45671=$b267      $3210=%0011 0010 0001 0000 %0110 1100=$006c
```

::

Or do integer arithmetic:

```
Questions:   D87*$84=          $3456+D32756=              %0111 1101/D4=

Answers:     D87*$84=$2cdc     $3456+D32756=D 46154       %0111 1101/D4=D 31
```

::

If you prefer floating point:

```
Questions:   123.22*87=          78900/.122=          15.7+459.0=

Answers:     123.22*87= 10720.1400  78900/.122= 646721.312  15.7+459.0= 5474.70
```

::

We shift to ASCII conversions (# means number, @ means ASCII)

```
Questions:   D49=@       $5c=@        K=#@                 ?=#@

Answers:     D49=@ 1     $5c=@ \      K=#@ $4b D75         ?=#@ $3f D63
```

::

Want to know the graphics characters generated by PF and control keys?

```
D1=@ ▌SOH:^A (Home)          ▛▜ =#@ $81 D129   PF1
D2=@ ▬STX:^B (Run)           ▌▐ =#@ $82 D130   PF2
D3=@ ▟ ETX:^C (Stop)         ▙▟ =#@ $83 D131   PF3
```

How do you print those graphics characters? You don't; CALC does. You simply press the key to be defined.

::

```
34-433     34.56
45-1234   122.40
...up to 255....
1-2        28.10
Take Cash (50.00)
          ------
Total:    135.06
```

We often need to sum (add or subtract) columns of numbers, and sometimes the lists get long, whether we're adding hours on a job, a deposit slip for a bunch of checks, or the costs of materials on a project. CALC will column sum up to 255 entries in any stand-alone version of the Editor, as shown at left. It reads mED memory, not the screen. Yes, you may sum integers or use up to nine decimal places.

If you make a mistake, you may correct the error, sum again and then save the tape to disk or printer. CALC also reports the number of entries summed. If you count 41 checks, and CALC says it summed 42, somebody blew it, hmmm?

We thank Joe Bostic for a strong helping hand when we married CALC to BEDIT. If you want to use CALC with other editors, you may. We wrote an interrupt-driven version which works with the Waterloo mED, John Toebes' V1.3, or the Development editor. There's a third version, CALCShort, without column sum, which runs in the monitor or in the languages. All versions are on the ISPUG disk we announce this issue, with full instructions and a tutorial.

A COMPILER FOR APL WITH A TUTORIAL DISK

Last issue, Reg Beck discussed a direct definition APL compiler written by Ted Edwards of Capilano College. In his column, The APL Express, Reg printed a number of examples showing what the compiler does. We'd sum up the advantages of the compiler this way: 1) You spend your time defining what you want done; 2) the compiler labors at the nitty-gritty of writing the code to do it. It's a classic example of how to free people for creative work and let the computer concentrate on the trivia.

Because all of us are impressed with the approach, Reg has issued a disk which holds the Ted Edwards compiler (it's in the public domain) and a tutorial on how to use it. The disk holds a series of screens or lessons. You can page forward and back through them and practice on each lesson. The tutorial is fully interactive. If you blow it the first time, you can go back and try again; if you want to repeat an operation, you may.

Once you get the hang of how to enter definitions, the compiler writes actual APL code for you to implement the definition. If you know what you want to do, and don't like composing APL sonatas in Greek, the compiler is for you. The disk is available from ISPUG, PO Box 411, Hatteras, N.C. 27943 in either 4040 or 8050 format for $10 U.S. Title: APL: Williams Compiler/Tutorial. It's for Version 1.1 APL only.

T H E A P L E X P R E S S by REG BECK

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

We will begin this column by looking at an old, familiar friend of mathematics teachers--the Venn diagram. Such diagrams and the functions used to generate them in APL provide the basis for exploring several APL primitives. Venn diagrams are used in set theory. The universal set is shown as a region of a plane; within this region, sets are shown as areas. In the simple diagram at left, the universal set is represented by dots; a subset, by asterisks. Such diagrams make sets easier to grasp.

```
.....*.....
....***....
...*****...
..*******..
...*****...
....***....
.....*.....
```

Operations performed on arrays are converted into Venn diagrams using the function VENN. The necessary functions and arrays are defined below. VENN and WITH are given in direct definition, but are easily written in del form if you don't have a direct definition compiler working. A is an 8 by 30 boolean array in a useful pattern for Venn diagrams.

U, (an array of ones), represents the universal set. The array B is obtained by a single rotation of A. A and B overlap somewhat so that we can find intersec-

tions and unions using A and B. We can easily generate the empty set from NOT U (an array of zeros).

```
⎕IO←1                          ⍝WORKING IN INDEX ORIGIN 1
U←8 30⍴1                       ⍝AN 8 BY 30 ARRAY OF ONES
A←(0 1 2 3 3 2 1 0-9)⌽(5 7 9 11 11 9 7 5)∘.≥⍳30
B←⌽A
VENN:'.*'[1+1=⍵]
WITH:⍺,(8 10⍴' '),⍵
```

VENN uses indexing to replace zeros with dots and ones with stars. WITH lets us display two Venn diagrams across the screen or printed page. The result of logical operations performed on the boolean arrays are transformed to Venn diagrams and displayed. The logical operations are analogous to set operations.

```
(VENN U) WITH VENN ~U
******************************          ..............................
******************************          ..............................
******************************          ..............................
******************************          ..............................
******************************          ..............................
******************************          ..............................
******************************          ..............................
******************************          ..............................

(VENN A) WITH VENN B
.........*****.................          .................*****.........
.......*******.................          ...............*******.........
.....*********.................          .............*********.........
.....**********................          ............**********........
.....**********................          ............**********........
.....*********.................          .............*********.........
.......*******.................          ...............*******.........
.........*****.................          .................*****.........
```

In the following two diagrams, we display the operations of union and intersection. The first shows the union of A with B. The corresponding logical operation is A OR B, which selects elements lying in A or in B or in both. The second, the intersection of A with B, corresponds to the logical operation A AND B. Elements which are in A and also in B are selected. Try the examples following the diagrams:

```
(VENN A∨B) WITH VENN A∧B
.........*****..*****.........          ..............................
.......***************.......          ..............................
.....*******************.....          ..............**..............
.....*******************.....          ............****..............
.....*******************.....          ............****..............
.....*******************.....          ..............**..............
.......***************.......          ..............................
.........*****..*****.........          ..............................

(VENN ~A) WITH VENN ~B
(VENN A∨U) WITH VENN A∧U
VENN A∧~A
```

NOT A is analogous to the complement of A, all elements not in A. When these diagrams are on screen, dump them to an APL printer using SDUMP (a dump on the first ISPUG utility disk). Pin the diagrams on your wall or bulletin board.

The definition of A is a bit obscure. Examine the following code and the array generated by it:

```
⍳30
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
  30
(5 7 9 11 11 9 7 5)∘.≥⍳30
1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

The rows of the array are determined by a successive comparison of each element of the left argument with iota 30. Whenever the element is greater than or equal to each integer of iota 30, a 1 is generated, otherwise a zero is produced. All that remains to be done is to rotate or shift the rows so that the pattern in A is formed. We accomplish this with the dyadic rotate function. Examine the following examples:

```
X←4 4⍴1 2 3 4
X           1⍉X         ¯2⍉X        (1 2 3 4)⍉X  (1 ¯1 0 0)⍉X
1 2 3 4     2 3 4 1     3 4 1 2     2 3 4 1      2 3 4 1
1 2 3 4     2 3 4 1     3 4 1 2     3 4 1 2      4 1 2 3
1 2 3 4     2 3 4 1     3 4 1 2     4 1 2 3      1 2 3 4
1 2 3 4     2 3 4 1     3 4 1 2     1 2 3 4      1 2 3 4
```

If the vector left argument has a single element, the right argument is shifted left by the indicated amount if the element is positive and right if the element is negative. To shift each row of the matrix, the left argument must have the same number of elements as there are rows in the right argument with the form of dyadic rotate used.

We can easily work out the vector by which each row of the array, Z, generated above, can be shifted to obtain a truncated diamond pattern. We can obtain the array, A, by shifting this pattern 9 places to the right.

```
(0 1 2 3 3 2 1 0)⍉(5 7 9 11 11 9 7 5)∘.≥⍳30
1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

For those without a direct definition compiler the functions VENN and WITH in del form are:

```
      ∇Z←VENN R                        ∇Z← L WITH R
[  1]   Z←'.*'[1+1=R]∇          [  1]   Z←L,(8 10⍴' '),R∇
```

Unconditional functions convert rather easily.

* * *

Paul Matzke of Madison, Wisc. pointed out an error in the function LEON in II, p. 37. Line 2 truncates instead of rounds; a .5+ was omitted. The correct version follows:

```
G←(⌊(.5+(E+.×TT)×100))÷100        ⍝CORRECTION TO ORIGINAL
G←.01×⌊.5+(E+.×TT)×100            ⍝AN IMPROVED VERSION
```

Thanks for the letter, Paul; I wish a few of the rest of the readers out there would write. Don't wait until you find an error in the column; write about anything which interests you in APL or on any new developments you may hear about. If want to see material which interests you, tell me what it is. WRITE!

* * *

Here in British Columbia, teachers who use APL are definitely in a minority. It is a hard, uphill battle to promote APL in the schools. I believe this is generally the case across North America. I continually see material in BYTE and other computer magazines written by scientists and engineers who extol the virtues of APL for their uses as superior by far to other languages.

Few of my computing science students say they will choose computing as a career; many will enter engineering or other technical areas. These students should be exposed to APL as well as to other languages. They should leave school at least with the knowledge that APL exists and could be of advantage in their careers. Perhaps the best place to promote APL is in the mathematics or science classroom since computing science is so Pascal-oriented.

I recently received the proposed curriculum guide for our newly-developed Computing Science 12 curriculum here in BC, which is to be totally in Pascal, to the exclusion of all other languages. Few secondary school computing teachers here in BC know Pascal. Most of them now teach BASIC and must learn Pascal. A recent resource manual, published by the Ministry of Education, is mostly in Pascal. I think the approach detrimental to the growth of computing in education, for if teachers can only use Pascal in the classroom, there will be no incentive for them to learn and pass on other languages and the evolving techniques which go go with them.

* * *

In my first column (II,#1, p. 21,22) a program was given for entry, sort and printout of a list of names. I took another look at the entry program and then thought of the recursive function MAT (p. 83,II,3) written in direct defiition by T. Traberman and presented in the last column. ENTER and SORT may be rewritten quickly in direct definition. I/O functions such as PRINT cannot be directly defined since they do not return a result. The rewritten functions follow. Compare them to the previous set.

```
ENTER:(ω↑V),[1]ENTER ω:0=⍴V←⍞:(0,ω)⍴''
SORT:ω[⍋ω;]
```

```
      ∇PRINT[□]∇
[  0]    PRINT A
[  1]   'IEEE4' □CREATE 1
[  2]   A □PUT 1
[  3]   □UNTIE 1
```

To use the program, return PRINT SORT ENTER 20 or replace '20' with the length of the longest name in your list. Type each name and <RETURN> after each entry. When the list has been typed in, <RETURN> an empty line and the program will do the rest.

* * *

If you're working in APL and want to print some ordinary u/l case text with the APL program listings, the procedure is quite simple. I have a workspace which contains FDUMP (see I, #15, p. 273), SDUMP (I, #8, p. 109), DTODISK and FETCH (II, #2, p. 37), the APL character set and two functions, TXT and APL. These functions convert the character set displayed on the screen to text or to APL. Either can be entered and returned, which permits switching between the fonts at will. The other functions let me do function dumps, screen dumps and send text to disk as screens. The text is screen dumped with the FX80 set up to print normal characters. To print APL characters, I download the APL set first. The dump functions and the APL character set will vary from printer to printer, but TXT and APL are universal.

```
      ∇TXT[□]∇
[  0]    TXT
[  1]   □AV[□IO+13] □POKE 59468
      ∇APL[□]∇
[  0]    APL
[  1]   □AV[□IO+11] □POKE 59468
```

At Dick's request I have put Ted Edward's direct function definition compiler on a disk along with tutorial lessons. The disk is available from ISPUG. [Ed. See separate announcement this issue. Reg did a fine job.]

EVALUATION OF WATCOM 6502 DEVELOPMENT SYSTEM

We asked Terry Peterson and Delton B. Richardson, who have used the 6502 Development System extensively, to report what they think of it. Those who are familiar with the 6809 Development Package (Editor, Assembler, and Linker) will find the 6502 package very similar. It is available from WATCOM Systems, 415 Phillip St., Waterloo, Ontario, Candada N2L 3X2, for $250 U.S. in the U.S. or Canadian in Canada. Delton reports first:

"This product at $250 is more expensive than other assemblers I considered, but it is an excellent choice for SuperPET owners. In operation, it is almost identical to the 6809 assembler that comes with the SuperPET. It runs in 6809 mode and uses the familiar microEditor, with minor additions such as a 'dos' command which works like our 'g ieee8-15.'

"It is first a very good and reliable assembler, supporting macros, structured programming statements, and bank-switching when you load and run 6502 programs in bank-switched RAM. It comes with a program disk, tutorial, and reference manual, and a software 'key' (or dongle) which must be inserted in cassette port 2 to run the assembler. The reference manual is adequate, but you will need other references if you aren't familiar with the 6502 processor. I'd suggest 'Program-

ming the PET/CBM' by Raeto Collins West as the best overall. WATCOM's tutorial is quite good; its examples of how to use the PET screen, keyboard, disk files, and printer are very helpful. The System can also be used to develop programs for the 8032, 4032, VIC 20 and Commodore 64.

"I used the assembler extensively over a period of two months to develop about 4K bytes of assembly code. When finished, I linked 14 separate object modules, representing a total of about 2600 source lines, didn't find even one bug in the operation of the system, and never reached any limits to its internal tables or memory structures. Overall, it is a superb product, easy, productive and reliable to use; I would recommend it to anyone."

Terry Peterson reports that an early version of the System had a few bugs; he now has the latest version, and reports these old bugs have been squashed. One minor problem remains, however: "The FCB pseduo-op fails to indicate 'LO' in the .b02 file for label references which do not specify 'HI' or 'LO'--causing the linker to add the label's high byte into the suceeding argument! Technically, I suppose, it should reject external label references that don't specify which byte to use; but I would prefer it to default to 'LO'."

Terry also reports that the latest 6502 System package incorporates the V2.0 microEditor, with a number of new features: line split, line join, a fill routine which pulls a line together after words or phrases are deleted, etc. One unhappy feature is that the commands on the shifted keypad (PF keys) have been changed; PF0, for example, now toggles between screen and command modes.

MicroEDITOR MANUAL FROM WATCOM Since Howard Sams ceased printing the Systems Overview manual, students and new users have no reference manual for SuperPET's microEditor. WATCOM has therefore published one, "Waterloo microEditor V1.1 - User's Guide for the Commodore SuperPET." The cost of the manual ($20 U.S. or Canadian in Canada) is higher than most other WATCOM publications because of the small quantities WATCOM printed. We received a copy of the 59-page manual, which is indexed and most complete. Order copies from WATCOM Systems, 415 Phillip St., Waterloo, Ontario, Canada N2L 3X2.

The manual is far better than the terse section on the mED in the Systems Overview manual. After three years of using mED, we learned a few things which were previously obscure. With the Systems Overview manual no longer available, this is a prime reference on how to use SuperPET's powerful microEDITOR, employed in all languages but APL to create and to edit programs. We recommend it.

SEARCH AND REPLACE IN THE MICROEDITOR
Part II

Last issue, we tried to define this process; we concluded with some rather simple search/replace commands. We now continue with more complex commands, some useful, some dangerous. You'll find three major types of what Joe Bostic calls "search/do" commands; one for Insert, one for Change, and one for Delete. All have a global search preface in the form of: */search-phrase/do, as shown below:

*/;/i Automatically finds the first line in text containing a semicolon and puts the mED into "insert" or "input" mode, opening up a new line immediately below the semicolon. You may then enter one or more text lines, go to the next semicolon by entering: . <RETURN> or with any function key; and there open new lines. The process continues to end of text. You get out of it only by standing on the STOP key until the screen flickers.

*/%^%$/d Automatically finds all blank lines in text and deletes them. The null is implied between %^ (start of line) and %$ (end of line), but cannot be directly stated in a search string.

*/%./c/%./!%& On any line which contains a character, change the first character to an exclamation point followed by the first character. The meta "%&" repeats the search phrase (here, the first character found).

The Change command (last above) may be confusing until you parse out the preliminary "search/do" phrase: */%./c --which is then followed with a normal search/change sequence.

Unfortunately, the command form above will not accept specific line ranges, as in: 1,40/;/d. Our new Editor, BEDIT, will accept such ranges.

Next, we come to a combination form, in which operations are performed on or between search strings. Suppose we have a file with numbered lines, as shown at left. These are not relative line numbers as mED counts them, but literally numbered lines in a file. We may delete these boundary lines and all between them with: /120/,/240/d. Similarly, we may go into insert mode at line 120 with: /120/i; we are given a blank line immediately after line 120. You might expect to use the change command with this form also. Suppose we have the code at left and we want to indent text between "loop" and "endloop". We try the next command, and it indents only line 510:

```
120 ...
... ...
240 ...
```

```
500  loop
510  if search$                    /510/,/600/ c/ /   /
...  ...
600  endif
610  endloop
```

So, we get tricky, and say: /510/,+2 c/ / /

It works. All code in the range is indented two more spaces. If you receive or write structured code which isn't clearly indented, this is a neat, fast way to solve the problem.

Next, we find some commands which are tricky or dangerous. Search with either of the commands at left, which simply say to find the start or end of a line. Then put the cursor anywhere, and say: //d . A //, of course, stands for the last search string given; you'd think the command would delete the current line. Instead, it deletes the first line of your file. If you try //i, it inserts a line after the first line in the file. Hmmm??

```
/%^/
/%$/
```

*/%^/d
*/%$/d You might think a null is implied in both commands at left, as it is in a previous example [*/%^%$/d, above, deletes all null lines in a file]. Alas, both commands at left delete an entire file.

*//d If you have specified a previous search string (such as "DDD"), you delete every line holding that string. No, the // does not specify a null unless you've been able to trick mED into making a null search string. We've succeeded a couple of times, but never did remember how we did it.

*//i Suppose your search string is a space; the command at left says to insert a blank line after all lines which hold a space. Sorry, Charley, but this command forms tiger stripes; it inserts a blank line alternately between single lines, between double lines...ad infinitum. Quick and dirty double spacing of text lines we don't get. Hmm. Try again:

*/%./i Will this give us a blank line after every line of text? Nope. We get tiger stripes again... Try it on a long text, with paragraphs of more than one line. No, */ /i doesn't work any better.

*/%./d This one says to delete any line containing a character, and that is exactly what it does--but it leaves all blank lines in the file. Suppose you want an accurate count of non-blank lines.... Hmmm.

*/%./c/a/b This command will indeed change the first "a" to a "b" on every line with characters on it--but it reports "Not Found". If you believe that message, you're in for trouble. Try it with a dense, long text file.

/ Enter this and <RETURN> it at command cursor. Does the search string become a null? If not, why not? Then enter a search string for a space, which is simply: / /, RETURN it, and try the next command:

*//c// / You expect any line containing a space will have its first null replaced by a space? Or does the second // search string also mean a space, and not a null? Try it. This is the hardest-working command alive; it grinds away and does absolutely nothing--except define what // means.

We've concluded that we'll never, ever see a complex set of search/do or search replace commands which are predictable and bug-free; the combinations are infinite; full testing and debugging aren't possible.

You are well advised to file whatever you're working on before using untested search/replace or search/do commands. As we've shown, they often do not work as expected. The result can be a disaster unless you're cautious.

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

This is the third 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. Don't assume that the abbreviation FCB means "form constant byte" in the text below; John deals with I/O routines; there, FCB stands for File Control Block.

SGETCHR_ : Read a character from the serial port : at $B099; JT $D5BB.

P1 - Address of FCB (File Control Block) opened to the serial port.
Returns - character input from serial port; unpredictable in case of error.

This routine is used to input a character from the serial port. It performs status checking on the serial port in addition to time-out checking to ensure that it is safe to get a character. It also verifies status flags in the FCB to ensure that no previous errors have occurred. In case of a time-out on the serial port, the system error message is updated and the appropriate flags in the FCB are set.

```
Example - CALL  OPENF_,#SERIAL,#READ
          STD   SIOFCB
          CALL  SGETCHR_,SIOFCB ;get the next char
              ...
          SERIAL FCS   'SERIAL'
```

Routine is used by the host communications routines to transfer files. It also depends upon the address of the serial port being filled

```
READ   FCS   'R'                          in the FCB.
SIOFCB RMB   2
```

SYSIOINI_ : System-dependent initialization for I/O : at $B0A8; JT $C1F5

Pass no parameters; returns nothing useful.

This routine is a very low-level system initialization routine. It empties the FCB area ($0580 to $0600) and configures the serial port at $EFF0 for 300 baud. It then sets up the host configuration which reconfigures the same serial port to 2400 baud. It also calls TIOINIT_ to set up the terminal, sets up the IEEE port, and finally initializes internal system variables. It seems of limited use in a program but may have application when you want to ensure that SuperPET communications are in a completely safe state. Calling this routine destroys all File Control Blocks. If you fail to reinitialize the FCBs with a call to routine INITSTD_, SuperPET will crash when it attempts I/O.

From the workings of this routine, it appears that an early version of SuperPET had more than one port to perform its serial and host communications. This routine attempts to pass a baud rate of 9600 to TIOINIT_, which completely ignores it. This routine is used by the main initialization code for SuperPET.

```
Example -   CALL   SYSIOINI_
            CALL   INITSTD_
```

REQUEST_ : Perform a specified process function : At $B0F0; JT $C1E1.

P1 - function requested.
P2, P3 - parameters (as appropriate) to the function requested.
Returns whatever function requested returns.

This routine is an attempt to provide a machine-independent implementation of exactly what the SWI instruction on the 6809 is designed for. When you call REQUEST_, the routine (or service) that you want performed is at a memory location unknown to the program that needs the service. By using REQUEST_, the code in the microEDITOR may be identical for all languages while the internal representation in each of the languages is different.

To use this routine, first stack the parameters for the function you call. Then load the number of the function (see below) requested and call REQUEST_. It will in turn invoke the routine whose address is stored at PROCESS_ ($002A) to process the request. The microEDITOR uses only four different requests although Waterloo defines nine different request functions, as shown at left.

```
QUIT    = 0   ;Return to Command Processor/Menu.
INIT    = 1   ;Initialize the Language/editor
EDIT    = 2   ;Invoke the editor
EXEC    = 3   ;Execute the Command Processor
ENCODE  = 4   ;Encode a source line
DECODE  = 5   ;Decode a source line
ALLOC   = 6   ;Add a source line
DEALLOC = 7   ;Delete a source line
IDENT   = 8   ;Identify yourself
```

These functions, although predefined, are available only to a program which is set up to support them. If you call REQUEST_ from a program that has not set up a procedure to handle it, then you will most likely end up in never-never land. When you load a program from the main menu, the SuperPET sets PROCESS_ to point

to the start of the code. It is for this reason that simply jumping to a routine already loaded in the banks will not work.

An interesting note about Waterloo's use of REQUEST_: although they take great care to preserve the parameters through several levels of calling after the initial request, the final routines ignore the parameters and use some defaults which happen to be the same as the parameters. One of these is DECODE, which always decodes the current line into the buffer at $0400.

```
Example - CALL  REQUEST_,#3 ;run the program
          ;now decode the current line to $0400
          CALL  REQUEST_,#5,#CURRLINE_,#LINBUF_
          CALL  REQUEST_,#1 ;reinitialize
```

TIMEOUT_ : Set device-specific time-out interval. At $B105; JT $B5BD.

P1 - FCB of file to set time-out for.
P2 - Time-out interval in seconds (approximate).
Returns time-interval set.

This routine sets the timeout interval for a device, indicating the maximum amount of time that SuperPET will wait before generating a TIME-OUT error message. All devices may have a time-out value associated with them although it is meaningless for devices TERMINAL and KEYBOARD.

```
EXAMPLE - CALL  OPENF_,#SERIAL,#READ
          STD   SIOFCB
          ;make the serial port time out
          ;after ten seconds
          CALL  TIMEOUT_,SIOFCB,#10
             ...
          SERIAL  FCS   'SERIAL'
          READ    FCS   'R'
          SIOFCB  RMB   2
```

The timeout value is checked in all of the major routines which perform I/O. Since the loops are hand-coded with internal constants to fix the repetitions of the loop to make a second, the timeout value is in fact approximate.

SYSREAD_ : Perform a system-dependent read. At $B108; JT $C2D8.

P1 - address of FCB for file to read from.
P2 - address of buffer to hold characters read in.
P3 - maximum number of characters to read.
Returns number of characters actually read in; unpredictable in case of error.

This is the main routine which figures out what type of I/O is to be performed on a file. It verifies that the file is readable, determines what type of device is being read from, and calls the appropriate routine to perform the actual I/O. It also maintains the File Control Block information, which shows error status, record position, and other miscellaneous file information appropriate to the type of file being read from.

```
Example -  CALL  OPENF_,#FILE,#READ
           STD   FILEFCB
           ;read 80 characters from the file
           CALL  SYSREAD_,FILEFCB,#BUFFER,#80
           STD   NUMREAD
           CALL  ERROR_,FILEFCB
                   ...
```

This routine is called by all of other I/O routines which perform file input. It is very similar to the standard UNIV V7 "read" function, which takes similar parameters. Internally, it does a lot of checking to verify that

```
FILE      FCS   'DISK.MYFILE'
READ      FCS   'R'
FILEFCB   RMB   2
BUFFER    RMB   80
NUMREAD   RMB   2
```

the operation can be performed correctly.

SYSWRITE_ : Perform a system-dependent write. At $B10B; JT $C52C.

P1 - Address of FCB of file to write to.
P2 - Address of buffer of characters to write out.
P3 - Number of characters to write.
Does not return anything useful.

This is the main routine which performs output on a file. It verifies that the file is writeable by checking both the access mode and the status flags in the FCB. It maintains the information about the file status in the FCB, including the current record position and any errors encountered. It is the main routine which determines which of the lower-level routines should be called to perform actual I/O. If in the course of output, an attempt is made to write a record longer than a record length specified in the file name passed to OPENF_, this routine sets the error code to 3 with the message, "Truncated".

Example -
```
CALL  OPENF_,#FILE,#WRITE
STD   FILEFCB
;write 80 characters to the file
CALL  SYSWRITE_,FILEFCB,#BUFFER,#80
CALL  ERROR_,FILEFCB
        ...
FILE      FCS   'DISK.MYFILE'
WRITE     FCS   'W'
FILEFCB   RMB   2
BUFFER    RMB   80
```

Internally, this routine is very similar to the UNIX V7 procedure WRITE. It may be called directly from a user procedure without any problems.

SYSNL_ : Perform a system-dependent newline. At $B10E; JT $C6C6.

P1 - Address of FCB to send a new line to
P2 - Flag indicates whether or not to reset the error status flag
Returns new file status - 1 = successful completion, 3 = error

This routine is used to perform a new line function on a file. It is called by all the system routines. It maintains the FCB information and resets any pending error flags if P2 is non-zero. In performing the new-line operation, lines in FIXED format files are padded to the appropriate length with spaces. It also sets the EOR flag in the FCB. Before operating, the routine checks to see if the file is indeed writeable.

This routine calls other routines specific to the device being written to which perform all the dirty work. When SYSNL_ is called after 80 characters have been written to the file TERMINAL, it does not create a new line, since the cursor is already on the next line.

Example -
```
CALL  OPENF_,#FILE,#WRITE
STD   FILEFCB
;output a new-line to the file
;remember to reset any error flags
CALL  SYSNL_,FILEFCB,#-1
CALL  ERROR_,FILEFCB
```

```
          ...
FILE      FCS   'DISK.MYFILE'
WRITE     FCS   'W'
FILEFCB   RMB   2
```

SOME BUGS IN WATERLOO COBOL
(and Some Random Nonsense)
by Fred Foldvary
1920 Cedar St., Berkeley CA 94709

Let's begin with a bug in the COBOL "THRU" statement; to understand it we must sketch in some COBOL background. In that language, programs are divided into four divisions; the DATA division describes the data used by the program and also provides names by which to refer to it. Within DATA, the "88 Level Data Items" (see par. 5.4.6 of the manual) allow you to assign a name to a given value of a variable.

For example, if you have a variable named TRANSACTION-CODE, you may assign the name ADD-RECORD to a value of zero, DELETE-RECORD to a value of 1, and CHANGE-RECORD to a value of 2. Thus, instead of writing "if TRANSACTION-CODE = 0", you can say "if ADD-RECORD" and have a more readable program, besides being able to change the values at the DATA division level while leaving the rest of the program alone.

The 88 Level also lets you give a name to a range of values, such as 8 through 90, which is equivalent to testing TRANSACTION-CODE for values of 8 through 90, inclusive. An example is shown below. The problem is, it doesn't work! The upper range works OK; values above the upper range test properly. But the lower range value does not. If TRANSACTION-CODE has a value of 5 in the above example and you test ADD-RECORD, the result is "true"--as if the value were 8 through 90! Values within the range defined test "true" as they should. For values less than the lower range, the THRU statement gives a wrong result in Waterloo microCOBOL.

```
01   TRANSACTION-CODE  pic   99.
     88  ADD-RECORD          value 8 thru 90.
```

Channel Problems Using Relative Files in COBOL: In the Waterloo system, one should be able to run a program using three relative files. A COBOL program using three relative files will run OK--if you only use three files. If you close one of the files and then open a new relative file, the program crashes. COBOL dies. The screen freezes. All data in memory is kaput.

Other circumstances can lead to similar effects. Using three sequential files and one relative file will lead to a crash if a file is closed and another opened. It seems that there is some kind of channel problem, where channels are not getting cleared when a file is closed. The problem does not occur at the opening of the new file, but in any I/O after the open.

I haven't tried other languages to see if the problem is system-wide or confined only to COBOL.

COBOL Program-ID Names: Our naming conventions call for COBOL program names to be appended with :c. That's fine for disk files, but the suffix must not be in the program name of the program-id statement, which the Waterloo manual says is for "documentation purposes only." Ha! The interpreter interprets the name you use; if you include a colon (:), the interpreter tells you that's an error. Any name must conform to the COBOL data name rules, which don't permit a colon.

I use the same name on my program-id as the file name, but stripped of the :c suffix. [Ed. mFORTRAN and mBASIC have the same problem; it's easily evaded by hiding the full filename as a comment. How often, O Lord, have we gone to refile an amended program, had the mED says the filename is "ieee4" after we made hard copy, and then had to figure out what the old filename is. We strongly recommend that full filenames be included in all programs--somehow--even if the name within the language must be truncated as Fred does it.]

* * *

Random Text, or "You Too Can Train Your Computer to Write Nonsense!" When I offered to share my random text program in the Gazette, our editor said there were articles about it in Scientific American and BYTE. Well, I have not been reading those magazines, but re-invented the wheel and wrote my own. Program "random_text" creates random words, with extra vowels to simulate their usage in English. So the text is semi-random, giving the flavor of English. Also generated are random-length sentences and semi-random punctuation, using periods, commas, and other punctuation based on random number ranges. You can vary the frequency of vowels, commas, question marks, and so on. The program, in microBASIC, also lets you input the output file names and the number and width of lines.

[We put Fred's short program on the ISPUG disk we announce this issue, and give you a flavor of results below. A recent book, "The Policeman's Beard is Half-Constructed," was written by a computer program named RACTER; some of the lines are haunting: "Reflections are images of tarnished aspirations"; some are a parody of intellectual chatter: "I myself am inflamed by Palestrina. Other countries besides Italy produced enrapturing composers in the 17th century; Seawell was an enraged, important Renaissance master. America was his nation. I take loving pleasure in his music." And sometimes the stuff is dull. Those interested should read "Computer Recreations" in the January '85 Scientific American, wherein the techniques involved are explained. For even more, read "A Travesty Generator for Micros", BYTE, p. 129, November 1984. If wordplay is your game, you'll enjoy both. Ed.]

Gs we xai. N. Racerta o jgs. Bicn meu.
Vaai. Ifugo j wa eaf ji? Wi ata na! Wago.
Uto ia le ia selo, mixo coejo eoge? Vsquaki.

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

215 Pemberton Drive, Pearl, Mississippi 39208

So far in our use of the Development System in SuperPET we've paid attention to the built-in Waterloo routines in the disk file: watlib.exp. We've also shown how to create a file of routines Waterloo did not document; we place such routines in a file called "usrlib.exp". As we see elsewhere this issue, there is yet another set of routines which we have until now ignored--those for doing floating point (or decimal) arithmetic. Though these routines are mentioned in the Development manual, the details of how to use them are not presented.

I'll attempt in this column to rectify the shortcoming with some examples. Before you proceed, please look at the article in this issue "JUDGE CRATER AND FPPLIB.EXP," which defines the purpose of all known floating point ROM routines.

As with all routines in watlib.exp and our own usrlib.exp, any routine in fpplib.exp is referenced in your program with an XREF in assembler files; in command files, the routines are made known to the linker with the line: 'include

disk/1."fpplib.exp"'. You use fpplib.exp exactly as you'd use any other of the export files.

The distinguishing feature of the FP math routines is that they use either one or two floating point accumulators (called FAC1 and FAC2) for almost all operations. If you deal with one value, you most probably will load the argument in FAC1 and then call a routine which expects a value in that accumulator. If you deal with two values, then one is usually loaded in FAC1 and the second in FAC2 before you process them. Some FP routines will load the accumulators for you.

The only catch is that the arguments in FAC1 and FAC2 must be in floating point format to be used by the FP routines. You must convert from string format (1234.56, as you enter it on the screen) to FP representation, using the program CNVS2F_. It converts a string with starting address P1 and an end address P2 into floating point, which is placed in FAC1. Be warned: the contents of FAC2 may be modified by CNVS2F_. Note also that the stop address (P2) of the string must be that of the character following the last digit in the decimal string to be converted. If, on the other hand, your decimal string is ended by a null byte you may use a dummy P2 (I suggest 0005). CNVS2F_ will stop converting when it finds a null marking the end of a string of decimal entries.

Once you have converted to FP format, you may need to convert another number from string to FP. If so, use the FSTORE_ routine, which moves the value in FAC1 to a safe location. You may now convert the second argument to FP in FAC1. You may avoid further movement of values by loading the first argument into FAC2 using function FLOAD2_; you're then ready to call whatever routine you wish to have SPET process the two numbers.

Now, unfortunately, the answer is found in a FP format (of which there are two; see John Toebes article on FP in this issue). You may want to decipher the number in FP format, but it is far easier to convert it to a decimal string by using CNVF2S_ [which is shorthand for Convert Floating Point To (2) String].

By now you probably wonder if you couldn't save time using an abacus or a calculator; you might wonder why the computer, designed as a number cruncher, must be so difficult to use for number crunching. An example, however, will reveal that what we have said above sounds much more difficult than it is.

It may seem a bit mad to you for me to now add 2 and 3 to illustrate the use of the FP routines (why didn't I use 2.22 and 3.33?), but as you'll later see, I must use integers here to later illustrate some of the other FP routines with this same program. Bear with me.

Let's address the problem of adding 2 to 3 [as the routine is written it'll work just as well to sum (get the total of) 2 + -3] Remember that the ADD routine has a sign (+), so that when we call it, we ADD any numbers we feed it, whether they are positive or negative. Here are the steps in the program below: 1) convert 2 to floating point format and store it; 2) convert 3 to FP format, 3) recall 2 in FP and add the two numbers, and 4) convert the result from FP to string format so we can read it.

```
xref cnvs2f_, fstore_, fload2_, cnvf2s_, fadd_  ; All are FP routines
```

```
; FIRST, convert 2 into floating point format in FAC1 and store it in a buffer.

ldd # end2    ; This is P2 for CNVS2F_, the end of the string + one byte.
pshs d
ldd # srt2    ; This is P1 for CNVS2F_, the starting address of the string.
jsr cnvs2f_   ; Convert from string value to floating point format in FAC1.
leas 2,s      ; Remove P2
ldd #buf1     ; This is the address where we store the converted 2.
jsr fstore_   ; We now have floating point form of 2 in our buffer 1.

; SECOND, convert 3 into floating point in FAC1.

ldd # end3    ; This is P2, the end of the string for 3.
pshs d
ldd # srt3    ; This is start address of 3
jsr cnvs2f_   ; Convert it from string to FP format and store it in FAC1
leas 2,s      ; Remove parm

; THIRD, get 2 into FAC2 and sum the two numbers.

ldd #buf1     ; P1 the address of FP representation of 2
jsr fload2_   ; Load FAC2 with the FP form of 2
jsr fadd_     ; Add the two numbers; result returns in FAC1; FAC2 destroyed.

; FOURTH, convert the result from floating point to string format.

ldd #buf2     ; We P1 the address of our buffer for the answer in string format.
jsr cnvf2s_   ; Convert the answer to string format; end it with a null.
swi

srt2 fcc "00002"
end2 equ *
srt3 fcc "00003"
end3 equ *
buf1 rmb 12   ; Since all string answers come back with a sign (or space), nine
buf2 rmb 12   ; digits, a decimal point, and an endstring null, 12 bytes are
buf3 rmb 12   ; normally needed to store a string answer.
end
```

```
"math"
org $1000
include "disk/0.fpplib.exp"
"math.b09"
```

If you care to, assemble and link the program, using the command file at left. Then test it. Use the .lst file; BUF2 is located at $1044. After the program runs, enter the monitor and dump the location with: >d 1044. You should see a 5.00000.

It is easy to revise the numbers added without reassembling and relinking the program above. After you "go" the program and get the first answer, leave the monitor. Get the file "math.lst" into mED, and find out the address of the numbers 2 and 3 (srt2 and srt3). So long as you confine yourself to no more than five digits, you can change the numbers in the monitor and "go" the program again (be sure to either leave the monitor between each "go", or to reset all 6809 registers to their normal starting values before you "go" again!).

The numbers will appear in the form of hex ASCII code, where 30 is zero, 39 is 9, 35 is 5, etc. To change our 2 to a 15.5 in "srt2", overtype the location

of "srt2" in the monitor like this (2e is "."): 00 31 35 2e 35 and then hit RETURN. If you now "go" the routine, you will add 15.5 to 3. You might change the 3 to a negative decimal fraction and add it. The hex ASCII value for a minus sign is 2d....

I'm sure none of us is startled by learning that 2+3=5, but perhaps we have learned something about how to use the Waterloo floating point routines. We'll continue to explore the workings of these routines in the next installment. In the meantime, congratulate yourself on not having to know how to form numbers in floating point format. But, if you're curious, see John Toebes' article in the next issue on how it's done. Until next time...

SPMON1 : A POWERFUL MONITOR The original version of SPMON was issued on the first ISPUG Utility Disk; since that time, Terry Peterson has substantially improved it and has added a small assembler and some other features. SPMON1 is much more powerful than SPET's built-in monitor. The following comparison of disassembled ROM code is worth ten thousand words:

Disassembly, Waterloo Monitor:

```
>t b21f.10
b21f  ROR  $f9,Y
b221  PULS $10
b223  PSHS $06
b225  PSHS $10
b227  TFR  $41
```

Disassembly from SPMON1:

```
d b21f.10
,B21F   66   39   |ROR   -$07,Y
,B221   35   10   |PULS  X
,B223   34   06   |PSHS  B,A
,B225   34   10   |PSHS  X
,B227   1F   41   |TFR   S,X
```

Would you have known that the ROR offset was a negative number? Could you have figured out that PULS $10 really means PULS X? That PSHS $06 translates to PSHS B,A--or that TFR $41 really says TFR S,X? If the answer is yes, do you <u>enjoy</u> parsing the bits and translating the nitty-gritty to mnemonics?

Terry Peterson wrote the zero version of SPMON (SuperPET Super Monitor) in 1983; he has since modified and expanded it to the final version 1.1 now in hand. We issued V 0.9 on the first ISPUG utility disk with a terse set of instructions which probably confused those who attempted to use the program. Terry finished V1.1 SPMON in March of this year; we subsequently sat down and put on disk a <u>full</u> set of instructions in the form of a tutorial which required two solid weeks of work.

Within SPMON, you can execute any DOS command, including those for directories, do all integer math except division in binary, hex, or decimal or mixtures of those notations; AND, OR, or Exclusive OR any values; and load any ML module, (language, facility, or your own program) from disk, either at its normal location <u>or</u> anywhere else in memory you may choose. Having it there, you may:

TRANSFER it to another location, HUNT through it for a specific chunk of code [SPMON returns the address(es) if found], DISASSEMBLE it, COMPARE it with another piece of code [SPMON returns the addresses of all differences]; or re-ASSEMBLE all or any part of it.

Built into SPMON1 are two means to record results: 1) You may divert any output either to disk or to printer, which is splendid for long disassemblies or for comparisons of two programs in which SPMON finds many differences; 2) You may

dump any screen from line 1 to the cursor either to printer or to disk, under any filename you care to assign.

One feature we find priceless: You may set breakpoints to stop execution of code at the nth pass through a loop; n is under your control. In the example at left, we set a breakpoint to stop execution the sixth time we encounter the instruction at $5405. You may have up to four such breakpoints active at one time.

```
b $5405 6
```

In addition to the GO instruction which runs code, you have instructions which let you step through the code, instruction by instruction, at your own pace, watching the registers as you go. QUICKSTEP runs the code to specified breakpoints, and then proceeds instruction by instruction at a keypress. WALK moves instruction by instruction, displays the results of the last step, and shows you the next step, which it won't execute until you press a key. At every step, you see the registers. For debugging, these two modes cannot be improved upon.

The little touches impress us. Example: The Condition Code register may return in Waterloo's monitor as C4. Tell us quickly: which CC flags are set and which are clear? At left is a register dump from SPMON; it tells you which CC flags are set; you needn't parse the bits.

```
r
  PC   A B  XR   YR   UP   SP  DP CC:bits set
;6039 0000 0000 0000 0000 0220 00 C4:efz
```

No program is of much use without good instructions, though; we pride ourselves that we wrote a pretty good set, in detail, and with plenty of examples, this time around. You can read the instructions in any stand-alone Editor, go immediately into SPMON, try the examples, and instantly return to the text in the Editor. SPMON may be used alone at main menu, in Development, with any stand-alone version of the Editor, or with microBASIC.

Old assembly language hands will find V1.1 of SPMON a delight; those learning assembly language will find it a much more powerful aid than the Waterloo monitor. SPMON comes in two versions, one which loads high and one which loads low in user memory; they work the same way. Both versions plus the full instructions are on a separate SPMON disk issued by ISPUG. It is available in either 4040 or 8050 format from ISPUG, PO Box 411, Hatteras, N.C. 27943, for $30 U.S. ISPUG receives its normal $10 for disk, postage, and Gazette subsidy; Terry Peterson we reward with the rest (gee, that works out to $6.67 a year for skilled programming; on this income, Terry can retire to Zambia as a peon).

We suspect that if a program of this quality were sold for the IBM PC, you'd pay one hundred bucks or more for it and consider it a bargain.

A PRINT-USING PROCEDURE TO FORMAT DECIMAL ENTRIES IN MICROBASIC

We've often wished mBASIC incorporated the PRINTUSING statement to let us format decimal entries with decimal points aligned. When we got desperate for one, we dropped a note to Frank Brewster, who lives up in Penn's Woods; he sent us the basic routine, as improved by Associate Editor Loch Rose. We changed it so you may optionally incorporate commas.

1. Procedure print_using. This sets the field size (number of digits plus one for the decimal point, plus one for the sign), and decides whether or not you want commas in numbers. Once the field size is set, you cannot enter any

numbers which are longer. You may call the procedure at any time you want to set or reset the field size or the print-using format.

2. Function fnf$. This formats the numbers and prints 'em out, all decimals aligned, with the number of decimal places you specify and commas (1,000,000.00 for example) if you want them. Trailing blanks are filled out with zeros (.7 becomes .70, and nnnn. becomes nnnn.00 at two decimal places). Our ThinkJet printer, which claims 150 cps and really does 110, can't keep up with the program.

```
110 ! format_dec:bd, a demo on formatting numbers with print-using.
120 D$=chr$(10) : print chr$(12); : open #14, "ieee4", output
130
140 !                       INSTRUCTION SECTION
150 print "Enter format wanted: in '###,###,###.###'; each '#' stands for a"
160 print "digit; commas are included, and 3 decimal places are specified."
170 print "If you don't want commas or decimals, leave 'em out."
180 print "The maximum value handled in decimal is 999,999,998.99.";D$
190 call print_using

210 proc print_using                          ! Call this to set or reset format
220   linput "Enter # format; no quotes needed: ", format$
230   if idx(format$,",") then commas%=1
240   dot%=idx(format$,".")
250   decimals%=len(format$(dot%:len(format$)))-1
260   field%=len(format$)+1                                  ! Allow for sign
270 endproc

420 loop                              ! Practice input routine for demonstration
430   input 'Enter a decimal number: ',nbr
440   while nbr>=999999999
450      input "Error. Number too large. Reenter: ", nbr
460   endloop
470   whatsit$=fnf$(nbr) : print whatsit$ : print #14, whatsit$
480 endloop      ! You must assign the function to a variable if you print
490 stop         ! to printer AND screen, or you call fnf$ twice.

510 def fnf$(fquan)                           ! The beast which does the work
520 ! If used without DECIMAL point, converts to CLOSEST integer value,
530 ! not to the next lowest integer; i.e., -.512 becomes -1
540  junk=int(abs(fquan)*10^decimals%+0.5)/10^decimals%
550  if fquan<0 then junk=-junk
560  flead$=""                        ! Zero "flead", or a NEG followed by a
570  if junk<=-1                      ! POS will print NEG!
580    flead$='-'                     ! flead$=prefix wanted in final result
590  elseif junk>-1 and junk<0
600    flead$='-0'
610  elseif junk=0                    ! Essential if somebody enters "000.00"
620    junk=0 : flead$=''
630  elseif junk>=0 and junk<1
640    flead$='0'                     ! Change to '+0' if + sign wanted
650  endif
660  junk$=value$(abs(junk))          ! Don't add flead$ yet! More work to do.
670  if decimals%
680    flen%=len(junk$) : dot%=idx(junk$,'.')
```

```
690     if dot%=0                                   ! Puts on required trailing
700       junk$=junk$+'.'+rpt$('0',decimals%)       ! decimals if some are entered,
710     elseif flen%-dot%<decimals%                 ! or if none are entered.
720       junk$=junk$+rpt$('0',decimals%-flen%+dot%)
730     endif
740   endif
750   if commas%                                    ! Adds commas if wanted.
760     dot%=idx(junk$,".")
770     guess
780       if junk$(dot%-4:dot%-4)="" then quit      ! Gee, we can squeeze in
790       junk$(dot%-3:dot%-4)=","                  ! commas without overwriting
800       if junk$(dot%-7:dot%-7)="" then quit      ! numbers--if we use reverse
810       junk$(dot%-6:dot%-7)=","                  ! English on n:n. Great!
820     endguess
830   endif
840   junk$=flead$+junk$                            ! Now add flead$!
850   fnf$=rpt$(" ",field%-len(junk$))+junk$
860 fnend
```

If there's room on your paper, we suggest you use a larger field than you think you'll need. It's terribly embarrassing to be nine-tenths done and find the last value entered is too large for the field!

GRAPHING WITH MICROSPACES
Part II

Last issue, we explored graphs created by microspacing a printer in one dimension; this time we look at the problems you encounter when you plot at fine increments in two dimensions.

If you want a method which can be adapted to any printer, you must form a matrix of graph values, so that no negative linefeeds are required. Many printers have no platen or rollers to grasp the paper firmly, but instead use pinfeeds. While some of these will do a negative linefeed, even the best can't bring the paper back to a previous line with enough precision to plot a point accurately. You must therefore print each line of the graph, from the top down, including scale and scale values, before you print the next line. A matrix is the obvious way to go.

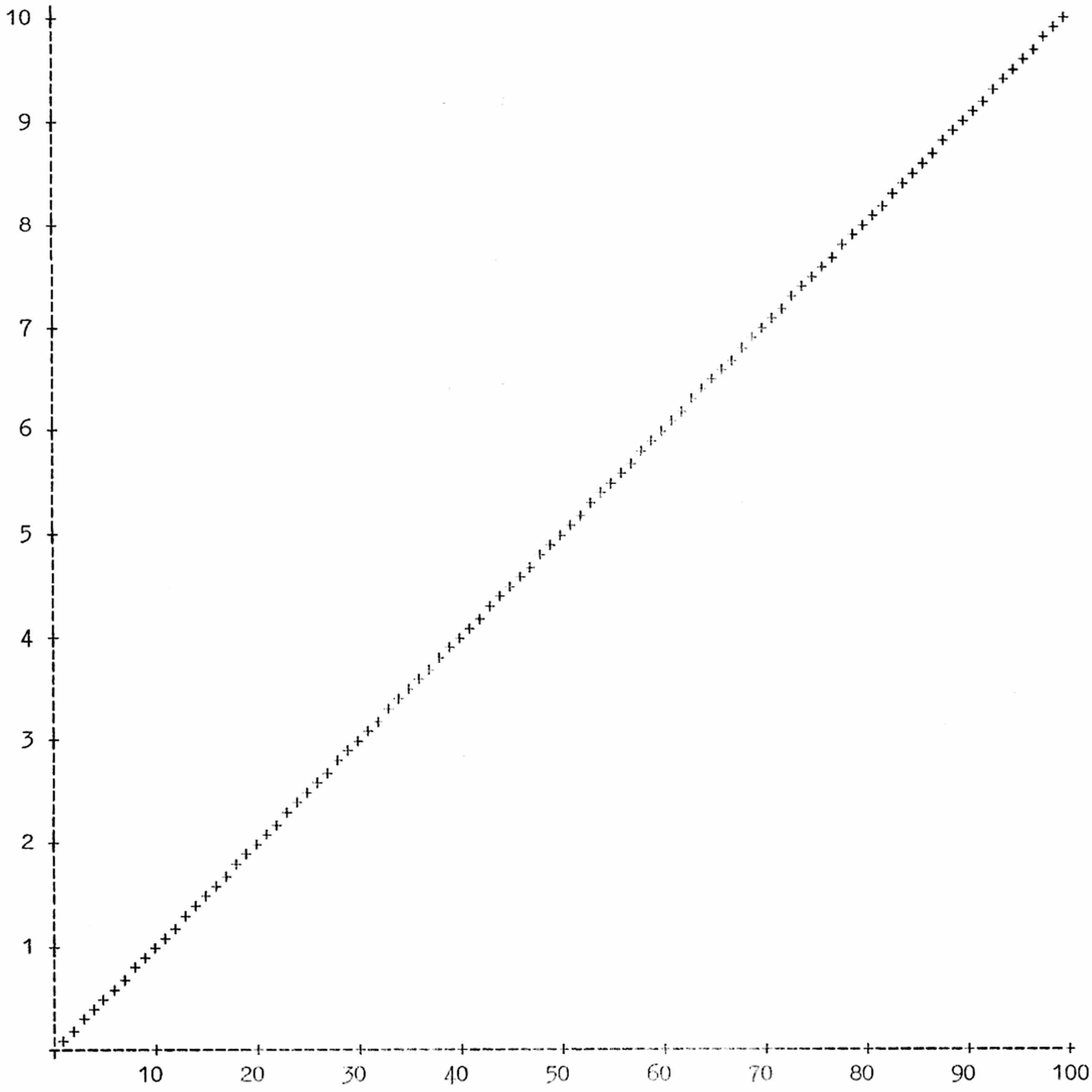
```
Row  1 |  x
Row  2 |    x
Row  3 |      x
Row  4 |        x
```

Unfortunately, a direct matrix approach won't work. The matrix becomes huge. Each row must be transformed into as many rows as there are vertical microspaces on that row. If there are six such microspaced rows for each line of a 50-line graph, the row dimension of the matrix becomes 300. If we intend to graph over the entire width of the page, the "column" dimension of the matrix may exceed 800. If we could confine the values in the matrix to one byte each, we'd still need 240,000 bytes to hold the matrix...

We can sneak around the problem if we recognize that the rows are the critical problem--and may compute the horizontal microspaces for each Microrow if the value to be plotted is stored in the matrix. We have now cut the matrix down to a tolerable size of 300 by 2.

Quite obviously we've ignored the possibility of graphing the values with bit-graphics because no daisy-wheeler can employ the method.

A STRAIGHT-LINE PLOT TO DEMONSTRATE LEVEL OF PRECISION



Line Plotted at .1 Increments on Ordinate
and Increments of 1.0 on Abscissa

Before anyone tries two-dimensional graphs, we strongly suggest he or she try a straight-line plot. It is easily done by forming a matrix which holds the coordinates of a straight line and by sending the matrix to printer with the printer set for the smallest possible vertical and horizontal microspacing. If the resulting plot is indeed a straight line or reasonably close to one, the work can proceed to more intricate shapes. If the printer mechanism is incapable of plotting a straight line, cease work. You'll never plot accurate curves.

The actual printing of two-dimensional graphs is simple. Two aspects of the job are more complex: 1) assigning data values to the proper row of the matrix, and 2) printing the scale borders and values. The third problem is a matter of bookkeeping. If you have microspaced over 722 microspaces in Microrow 1 to print a point, it makes little sense to do a CR and space over 725 microspaces in Microrow 2. Instead, keep account of the last position, and space right or left only the difference in values.

We show on a facing page an approximation of a straight line, done in the manner outlined above. It shows that our printer is capable of fairly good accurancy up to increments of .1 on the scale shown. The line degrades quickly at smaller increments. Knowing the printer limits, we can now assign the proper scales and plot values with knowledge of what the printer can and cannot do.

In sum, if your printer will microspace with reasonable accuracy, you may generate good one- and two-dimensional graphs in a relatively simple and painless way without leaving text mode and without bit-graphics, even on daisy-wheelers.

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	No. 4: $3.75

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

==========

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

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

Name:_____ Disk Drive: _____ Printer:_____

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

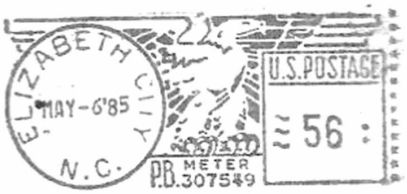
[] Check if you're renewing; clip and mail this form with address label, please.
If you send the address label or a copy, you needn't fill in the form above.

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

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

Table of Contents, Issue 4, Volume II

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



First-Class Mail
in U.S. and Canada