SuperPET Gazette

WATERLOO PASCAL and STRUCTURED BASIC are now available for the C-64. Hmmm. We got the red herring from WATCOM just before this issue went to the printer, and began to wonder how WATCOM crammed 'em into 64K with any room left over for programs, and about the implications... In SuperPET, the 64K of bank-switched memory holds the interpreters and the bank-switching routines (pure overhead). If we assume WATCOM optimized the code to condense it (and exclude the bank-switching), we figure the interpeters might go into 48K if handled as-is. Hmmm. That doesn't leave much for programs. So we read the fine print, and, lo! Structured BASIC is done in a 4K ROM, and clearly is similar to the version which runs on the 8032.

The PASCAL offered seems very similar to SuperPET's. The interpreter is on disk, the EDITOR in a 16K ROM (WATCOM picked up some space!). Programs are limited to 700-800 lines--and the EDITOR (which we suspect is a version of the mED) can be loaded alone. Prices: $149 for one copy of PASCAL; $99 for one of BASIC. Added copies, ordered by schools at the same time, are $99 for PASCAL, $79 for BASIC, in Canadian funds in Canada and U.S. funds elsewhere (20-25% surcharge on all furriners, please note). Write WATCOM Products, Inc., 415 Phillips St., Waterloo, Ontario, Canada N2L 3X2.

What does this have to do with SuperPET? (1) Our readers in schools deserve to know what is available; (2) the shape of the future doth appear. Commodore recently announced that the C-64 will be sold only to schools in an 8032 case with a green screen; we hear the price is about $500. Does this mean the death of the 8032 and SuperPET, which Commodore makes by adding one board to the 8032? Will WATCOM soon announce that FORTRAN is also available on the C-64? We ask these questions during a deadly shakeout for computer makers and software houses. Only a few computers and a few major software super-packages will survive. Forget the computers for a moment, and ask: "How how many software packages do we need?" Soon, the answer will be one integrated series, comprised of a word-processor, a spreadsheet, a data base/mail list manager, and an accounting package. When two or three such major SuperSoftware sets compete, who will buy any of the hundreds of separate programs which now flood the market? Software for teaching faces a hardware sieve; if such software doesn't run on an Apple, a PC, or a C-64, how much will you sell? Inevitably, schools will teach on machines made in quantity for commercial use <u>and</u> teaching. Will languages suited for commercial use <u>and</u> teaching dominate and destroy teaching languages? Given the choice, which would you buy? Are structure, good debuggers, and the ability to interpret as well as compile useful only in schools? Do commercial users want no structure, bad debuggers, and no way to check code before it's compiled? Watch the ongoing drama in the marketplace. Our questions will be answered there.

IF YOU'RE RED, YOU'RE DEAD

Look at the mailing label on this issue. If, <u>underlined in red</u>, you find a note that your membership has expired, <u>this is your last issue</u>. Please renew <u>now</u> if you're going to renew. Check the RENEW block on the last page and send it with your address label or a copy (you needn't fill the form if you send the label). Honor us with $15 in North America or $25 U.S. from overseas, checks to ISPUG.

ONCE OVER LIGHTLY **Miscellany** From Associate Editor Terry Peterson: "I've found out that A.B. Computers [252 Bethlehem Pike, Colmar PA 18915] has the SuperPET version of PAPERCLIP (9000A) for $60! Woe, I did not find this out until after shelling out $150 for it...Be Prepared to insist if you talk to A.B. I asked the salesman there about the SPET versions before I ordered elsewhere, and the response was "Duuh. Wazzat?" To my pocketbook's chagrin, I gave up." So, don't give up. Insist all $90 worth!

Also from Terry: "An undocumented feature of the 6502/6809 Waterloo assemblers: If you give the assemblers, in response to the filename prompt, a string prefixed with "-l" [Ed. The letter "l", folks.] the effect is similar to including an 'option nolist' within the .asm file. It's very useful." For the uninitiated--"option nolist" tells the assembler not to generate that long "list" (.lst) file during assembly.

HARD HARDWARE! Jakob Bennema of the Netherlands writes that while we gave a few good reasons for "going Commodore" in a recent issue, he has a better one. In late '82 he drove toward home with a new SuperPET and an 8050 drive, crashed, totalled the car, zapped himself, and threw SPET and drive out of their shipping cases. Damage to SPET and drive: none. Heckuva way to make a point, Jakob.

WHY THE *SEQ ON DIRECTORY COPIES? Ever send a copy of directory to disk in V1.1 with "di disk index" and wonder why the copy of directory you bring into mED shows a *SEQ after the filename "index"? The same file on directory shows without the asterisk. What goeth on? Gee, we finally figured it out: The file "index" is opened, and dutifully includes all files on directory, including--ho hum--a new file called "index", which <u>isn't</u> <u>yet</u> <u>closed</u>. Golly, if it isn't yet closed, it'll be *SEQ to show that fact. Then, of course, having properly reported itself out of order, it closes. And any subsequent directory call shows the file with no asterisk. Elemental logic, Watson. Only took us 18 months. Quiz question: Why doesn't this happen when an old file 'index' is on the directory?

AN INEXPENSIVE MODEM Bob Kobenter of Canada has gone to Siemens AG as a programmer, but reported as he left that the EMP MANUAL MINI-MODEM MM101, at $85 U.S., did a fine job for him at 300 baud with NEWTERM. Operation is manual, but he recommends it for those whose dollars won't stretch further.

THE PURLOINED CONTROLS Seems some of you didn't read Gary Ratliff on text compression in the mED (Issue 12) and still wonder why all CONTROLs (ASCII 1 to 31) are deleted from any file brought into mED, edited, and refiled. Well, the mED uses the CONTROLs in the text compression algorithm, so they cannot be in the text too. If you really want to keep CONTROLs after editing a file:

Translate all CONTROLs, either to their +128 counterparts in reverse field, or to symbols you usually don't use, such as backslash, tilde, accent (shifted @), or up-arrow. Do this in program, and put the translated file to disk. Then edit. When through, file to disk, and run a program to retranslate back to the original CONTROLs. Since the mED can't generate any reverse-field characters and you dassn't edit any line with such characters (it destroys them), the +128 counterpart option isn't good. Your best bet is to substitute the characters we note above, which you can enter and can edit in the mED. It's easy to write and run the translation programs, to and from. Be sure you don't try this in any version of the mED which runs in language, which will royally eff the files. Use the mED loaded alone, or the one which runs in DEVELOPMENT.

A WILTED ROSE We finally got V1.1 software to Loch Rose a month or so back. Poor fella had it on order from his dealer for two years, and, being delighted, tried V1.1 microBASIC on some old V1.0 programs--and wilted. These files, in PRG format, wouldn't run properly. Loch reports he salvaged them by getting them off disk in V1.0 with a LOAD (PRG format) and SAVEing them to disk (SEQ format); the

SEQ files, loaded in V1.1, run okay--unless you happen to employ one (of few) syntax differences which V1.1 doesn't like. We repeat, SEQ files created in V1.0 run okay in V1.1 and vice-versa. Once you get an old file into V1.1 and running, you can then STORE it as a PRG file. Be warned.

THOSE APL KEYBOARD DECALS The APL keyboard layout, made of thin decals atop cardboard, still lies above our main keyboard. George Parry writes that you can actually put the markers on the keys. He did it by removing the keys themselves with gentle upward pressure and a small tool, and then affixing decals. Note the decals don't go on the top of the keys, but on the front face--where they are effectively obscured. If you want them on anyhow, position them with a pair of eyebrow tweezers; press 'em into place with a Q-tip slipped between the keys. George reports the keys come off and go back on easily (they do indeed) if you don't lose the little spring underneath. You can't clear the keyboard contacts this way; a rubber diaphragm at the bottom of the key tube closes it completely. So leave the keys alone unless you're all thumbs or out of tweezers.

STOP DOES Loch Rose writes that he notices "a number of Gazette programs in mBASIC have 'reset : stop' on the last line. Since STOP closes all open files, the 'reset' is redundant, is it not?" It is. Sorry, Loch. We're conservative, always close files out of habit, favor stiff collars, hair sofas, Kipling, the gold standard, the flag, motherhood, a sound dollar, and vote straight Whig.

VALUES FOR SERVICE_ WATCOM has replied to a letter of inquiry about the various value for the system service variable SERVICE_ at $32, and to the left are those values as defined by Waterloo.

```
QUIT   =0 ;Return to Command Processor/Menu.
INIT   =1 ;Initialize
EDIT   =2 ;Editor Requested
EXEC   =3 ;Execution Requested
ENCODE =4 ;Encoding of Source Line Requested
DECODE =5 ;Decoding of Source Line Requested
ALLOC  =6 ;Add a Source Line
DEALLOC=7 ;Delete Source Line(s)
IDENT  =8 ;Identify Yourself
```

If any of you bit-twiddlers have further definitions or examples of general use to other bit-twiddlers, send 'em in.

SORTED MEMORY MAP DATA With much help, we've put together two sorted lists of system routines/pointers/addresses, one sorted by names of routines, and the second by address. The lists include routines from WATLIB.EXP, FPPLIB.EXP, those located by Ratliff, Larson, Toebes, Rose, Peterson and others, plus the undocumented integer arithmetic routines and a bunch of zero page pointers/addresses from WATCOM. Each list now exceeds six pages; you can find anything by its name or address. We've added the jump table addresses to each system routine. To get a copy (14+ pages), send $1.50 U.S. for printing and postage to Editor, Box 411, Hatteras, N.C. 27943. If you find errors or omissions, or can further define the routines/pointers followed by ??, tell us! Don't expect the Holy Grail.

WHAT'S INTEGER? Loch Rose has been waiting for three months for a printer, but can't get one delivered because of the worldwide shortage of chips--particularly those made by Intel. Undefeated, he sends his notes writ by hand. Latest: "Terry Peterson pointed out (I, 116) that the mBASIC function 'idx' returns an integer result, and discussed the consequences when you use the logical operators AND, OR, and NOT with this function. The manual does not say which functions return integer values, so I checked all of them. No less than nine functions can return integer values. These seven always return integer values: cursor(i%), peek(i%), idx(a$,b$), hex(s$), len(s$), ord(s$) and sgn(x). The other two, abs(x) and int(x), return an integer value if the parameter is in integer variable. Thus abs(i%) returns an integer; abs(1) and abs(x) do not. Oddly, int(i%) returns an integer, but ip(i%) does not." The summary is for mBASIC. The same AND, OR, and NOT problems exist in the other languages with integers. Any lists?

HOW TO GIVE GRANNY A FACELIFT WITH A CARVING KNIFE Well, it's not quite as dangerous, but Russ McMillan of Madison WI writes: "Here's how to rewrite a directory header on a 4040 disk and not lose the files: Put a disk with the right header in drive 0. 'Mount' this disk. Remove it and insert in drive 0 the disk whose header is to be changed. 'Put' an empty file to drive 0 (e.g.'p zilch'). You now find the disk in drive 0 has the header of the first as well as its BAM. Validate the disk to correct its BAM, and scratch file 'zilch'. You can guess how I discovered this." Yup. Tumble into the privy and leap out with a rose.

SUPERPET microFORTRAN in SCHOOL
A Software-Book Review
by Stephen Pace,
Snowflake High School, Box 1100
Snowflake, Arizona 85937

This semester, I have been using FORTRAN FOR STUDENTS, written by G.W. Booth, in my advanced programming classes. I have found the book to be easy to use on an individualized basis. Each lesson consists of written material and examples that are loaded from the example disk. This manner of presentation I found to be a good way for the students to work with new concepts. A student can modify an example as much as may be needed until the concept is mastered, unlike a standard textbook, where all they can do is look at it.

At first I encountered some resistance from my students, who were used to textbooks with pictures, charts and drawings. Now that the students have used the Booth text, most of them quite like it. They all feel that it is an improvement over the tutorial in the manual that came with the SuperPETs.

The first nine lessons in the text discuss the commands and operations of the microEDITOR. The last fifteen lessons include formatting, read statements, loop/endloop, until, while, if...elseif...else...endif structures, string operations, subroutines, sequential files, and relative files.

The problems that I have had with this book are few. I think some students would appreciate having an occasional 'fun' exercise like programming a game. Some of my students had a little bit of a problem with the section on commands at the beginning of book. Several of the commands seem to indicate that quotes are necessary when none are actually needed. I hope that an index can be included.

Overall I give the book a very favorable recommendation. The author may be contacted at Parkland Composite High School,4630 12th. Ave., Edson, Alberta, TOE OPO, Canada. [Ed. We chatted with Mr. Booth, the author, after this review came in. The book contains 25 lessons in 94 pages, plus 15 pages of prefatory matereial and overview. The author has added an index. Two disks, available in either 4040 or 8050 format, come with the book; the tutorial disk holds 46 programs tied to the book; the second disk, the answer set. The last lesson sets forth five major programming projects in which the student must employ what has been learned. The book and disks may be ordered directly from the author at the address above. For $500 Canadian, schools receive one copy of book and disks which may be copied as needed within that school. The permission to copy is realistic, for while Mr. Booth supervises 24 SuperPETs, we know of schools in which one teacher supervises 90; even at $10 per copy, the price would be prohibitive. Let it be clear that this book is designed specifically around SuperPET's mFORTRAN.]

FOR WANT OF A NAIL... ...the battle was lost, and for want of a ha'penny felt no larger than a pencil eraser, your disk drives will quit reading and writing. On 4040 and some 8050 drives, you can avoid the cost and inconvenience of some drive repair jobs.

A major cause of "IO/Time Out" messages is bad connections. Living near the sea, ye ed catches #$! from salt-mist corrosion. Whenever "IO/Time Out" appears, we immediately: (1) slide back and forth (wipe) all IEEE-488 connections. Usually this alone puts us back in business. If it doesn't, we (2) lift the lid on the drives and wipe every plug connector in the drive, running 'em back and forth at least ten times to get firm contact. That usually cures motors that won't run, R/W heads that refuse to work, and ends device not present messages. A very few times, we've had to move socketed chips up and down in the sockets.

The simple tricks above take care of about eight of ten disk drive failures. The rest of them have been traced to a problem we've never seen mentioned anywhere: badly worn pressure felts. What are they, and where? Something must press the disk itself downward until it touches or almost touches the read/write head--and that something is a tiny felt cylinder which rides on the top surface of the disk. As the felt wears, the downward pressure on the disk decreases until the read/write head no longer reads or writes reliably. The shiny sections you see on a well-used disk are buffed by the pressure felt.

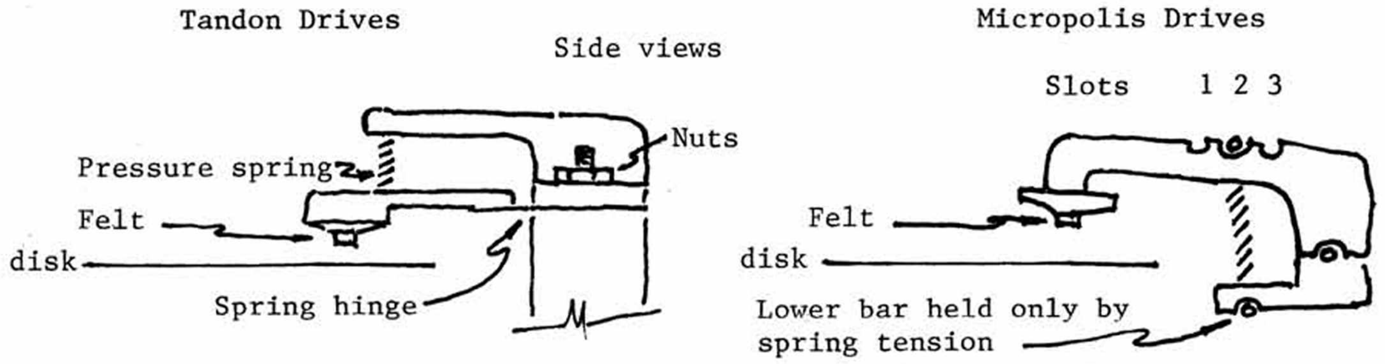
How long do the felts last? Under heavy use (6-8 hours a day, five days a week), for less than nine months. We have to replace ours at least that often. It would appear that some repair shops never look at them or replace them. Though Steve Zeller sent his 8050 out for a repair a few months ago, the felts weren't replaced, and Steve was soon out of business. We sent him spare felts. That solved his problem. He then recommended we write this article.

Steve found out he could get felts directly from Commodore. If you need them for Tandon drives, the part no. is 990011. Call Commodore at 215 431 9200 for the part number on Micropolis drives. Send checks to CBM, 1200 Wilson Drive, West Chester, PA 19380. The Tandon felts cost $1.30 each (buy a bunch whilst at it!). Tandon drives have a top-hinged door; so do (!*!) some Micropolis drives which confine the read-write head in a turtle shell of metal (we can't repair these). Other Micropolis drives use a push-down latch at the bottom of the door. These can be fixed. You can make your own felts if you find utterly clean, dense, instrument-grade felt which doesn't shed lint. Pads are about 0.83-inch high (just over 5/64-inch), including the paper which shields the adhesive on new felts. We have found that felts a bit overlong are okay (no mars on disk from high pressure against the R/W head). Use them at your own risk.

If you're game to replace the felts yourself, secure the rare items at the left.

Tools and Gear:
One hatpin (3-inch)
Some naptha or lighter fluid
One small dental mirror
One pair of needle-nose pliers
Two dozen fierce expletives

The dental mirror will be found at any good hardware store, and is the small, round mirror-on-a-handle with which your cavity-chaser peers at your wisdom teeth. Get a small one. You need naptha to clear the old adhesive from the felt holder. The pliers should have needle jaws some 1.25 inches long. Any hatpin suitable for the defense of female virtue, even if blunt (hatpin, not virtue), is fine. If you have a Micropolis drive, you'll need at least four paperclips in place of the hatpin. The expletives will be required in the quantity specified whatever the drive, for here is what you'll see:



Use the dental mirror before you do anything else. If you see 1/32-inch or less of felt below the plastic cup which holds it, your felts are worn. It's not uncommon to see no felt at all on a long-used drive. With the circuit board propped out of the way, see if light downward pressure, applied with a wooden tool against the felt holder to increase pressure, will let the drive read and write. You may increase the pad pressure on Micropolis drives by moving the top bar from Slot 3, its as-shipped position, to slots 2 or 1. But even this won't help if the pad is worn down to the nub. If increased pressure solves the problem, you know you need new felts.

Steps: After you open the drive and record on paper the circuit-board connections, (1) remove the plastic pressure head. On Tandon drives, you must remove 2 tiny nuts to do this. If you can find a socket to fit, fine; we couldn't, and so used needle-nose pliers. Note that the studs on which the nuts are threaded are hollow. Insert the hatpin into the hollow--and then remove the nuts, sliding them up the hatpin when they are free. You'll never find the nuts if they fall. On Micropolis drives, there are no nuts. Instead, you'll see a bar lying in a slot. Lift the bar, turn, and the pressure head will come free--as the lower bar at the other end of the spring, held in its slot by the spring, drops (ooooosh!) into the works.... Don't let it happen. Rig a Rube Goldberg line and hooks made from paperclips, to lift and hold the upper bar and spring after you free and remove the plastic pressure arm (the hook rig must come in two so you can get the pressure arm fully off--tricky!).

Step (2) is simple. Remove the old felt; clean off the old cement with naptha. Remove the protective paper from the pressure-sensitive adhesive, and insert the new felt firmly. (If you make your own felts, Devcon Rubber Adhesive, a clear yellow substance, holds like a mastiff once dry, but needs at least 30 minuntes to dry. You must put on two coats because the felt is so absorbent. If you get any adhesive either on the side or pressure-face of the felt, throw it away.)

Step (3) is a reverse of step (1), except that you must align the felt carefully over the Read/Write head on Tandon drives. Use the mirror. When aligned, tighten the nuts, and check again for alignment. Micropolis drives are self-aligning. When finished, check the rig with a scratch disk and a backup.

The sharp of eye wonder how you convert purchased felt to tiny cylinders. Easy. While you're buying the dental mirror, get a belt-hole punch--the kind with five or six hollow punches on a wheel, which will make five or six different sizes of hole in leather. Pick the punch to suit, and punch out a dozen or so felts. Trim off both ends of the cylinder, for we haven't seen a felt-seller yet who maintained a dust-free storage room. Department store felt won't work. You must have lint-free, clean, white instrument-grade felt. We got some 3/8-inch thick, which should last us until A.D. 2000 or so.

This sounds about as difficult as it is. Once you've gone through it, you'll be able to do it without a resupply of expletives. Be sure to return the hatpin, for you never know when virtue will require defense. Warning: Commodore has not approved the method, and you proceed at your own risk.

ON CONVERTING PAPERCLIP FILES TO 6809 ASCII FILES

Having been somewhat bewildered on how to convert 'CLIP files to ASCII files readable in the mED in 6809, and having found that others were equally confused by what the 'CLIP manual does not tell you, we outline below how to do it. (Read page 6.5 in the 'CLIP manual first.)

1. Load the ASCII printer version of 'CLIP.

2. Call the file to be converted to ASCII to screen.

3. Command: CONTROL # 8 <RETURN> [Nothing seems to happen]

4. Command: CONTROL o <RETURN> and then go through that long option list for the printer, accepting all of 'CLIP's default commands.

5. Eventually, the program will ask you for the filename and disk for the new file. Give them. The screen file goes to disk--and not to printer--as an ASCII SEQ file.

6. Leave 'CLIP. Load mED. Get the new file (use CAPS for the filename) and you will find perfect ASCII files in whatever format you set in 'CLIP.

ON FONT CHANGE, COMMODORE GRAPHICS, TEXT WITH GRAPHICS, and THUDS

Indeed you may decorate your programs with the Commodore graphics fonts from 6809 mode. (Some readers asked.) Shrewd heads will note the plural "fonts". We suggest you read pp. 68-69 of the System Overview manual before you go further into this article.

Before we go on, a warning. In those languages which have an immediate mode, in which you can PEEK and POKE out of program, font changes come easy. But don't draw any conclusions from such trials about what'll work in program. Font change is controlled at $E880-E881, the CRT controller. In immediate mode, you can POKE a value there and it will remain unchanged. This isn't true in program; to see what happens, run any program of length in any language, and stuff in some PEEKS of $E880-E881 here and there. Lo! The values change. To keep from crashing when you change between WATERLOO and COMMODORE fonts, always switch with the POKEs at the left. At times, you can switch back without a first POKE of 59520 and not crash, but then again sooner or later thee will. Use both POKEs at each switch between Waterloo and Commodore fonts while in program.

From Commodore to Waterloo	
poke 59520,12	or $E880,0C
poke 59521,48	or $E881,30

From Waterloo to Commodore	
poke 59520,12	or $E880,0C
poke 59521,16	or $E881,10

Once in the Waterloo or Commodore font-pair, you may choose either of two sub-fonts. When you complete either of the sets of POKEs at left, above, you obviously will get one of the sub-fonts by default. You select the sub-fonts with the POKEs shown at left, below. If you are in 6809 mode, be warned: 1) If you are in APL and POKE to Waterloo Roman, the keyboard will be a hybrid though the font will be Roman, and, 2) If you are in Roman and poke to APL, the keyboard again will be a different hybrid. If you're not going to use the keyboard in program, this is no problem. If you are, you may avoid the keyboard problem with a SYS, as noted on pages 115-116 of Issue 9.

For Waterloo Sub-Fonts	
poke 59468,12	Waterloo Roman
poke 59468,14	Waterloo APL

For Commodore Sub-Fonts	
poke 59468,12	Pure Graphics
poke 59468,14	TEXT, Graphics

Assume we've poked to the Commodore font from microBASIC. We were in the Waterloo Roman font, and the value of 12 (Poke 59468,12--see above) governs. We get the Commodore pure graphics font called by that value. You'll find only graphics characters on the keyboard. You aren't stuck there forever; carefully type: poke 59468,14, even though you can't read 'poke', and you'll switch to the Commodore text+graphics font. There, you'll find everything is turned around; graphics are on the unshifted keys, and characters on the shifted.

When ready to return to the Waterloo fonts, enter the right POKEs (left,above), and you'll come back--but in APL. Why? The Commodore text+graphics sub-font has the same value at 59468 (14) as APL does in 6809. The System Overview manual does not mention these across-font defaults. Obviously, in this case you'd want to poke 59468,12 to get back to the Waterloo Roman font. We demonstrate all this vividly in the short program below, which also shows how to eliminate spaces between text lines for good graphics.

```
100 ! "them.fonts:bd"
105 print chr$(12); : z=255
110 print " p@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@n"
115 print " ]                                                             ]"
120 print " ]           WATCH WHAT HAPPENS TO THIS BOX AND TEXT           ]"
125 print " ]                                                             ]"
130 print " ]                                                             ]"
135 print " m@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@}"
140 if cursor(z) then print "In Waterloo. Next: remove spaces " : call delay
145 call lines_out
150 if cursor(z) then print "Next: Shift to pure graphics     " : call delay
155 poke 59520,12
160 poke 59521,16                            ! Choose Commodore fonts.
165 call delay
170 poke 59468,14
175 if cursor(z) then print "IN TEXT/GRAPHICS. NEXT: TO WATERLOO" : call delay
180 poke 59468,12                            ! Return to lower case sub-font.
185 poke 59520,12                            ! Choose Waterloo fonts.
190 poke 59521,48
195 if cursor(z) then print "We're back. Next: get line spaces  " : call delay
200 call lines_in
205 if cursor(z) then print "  Back to normal line separation   "
210 x=cursor(1841) : stop
215 !
220 proc lines_out
225 data 4,40,5,5,7,33,9,7,0
230 loop
235   read i
```

****

It's quite easy to design your graphics in 6809 mode, using characters which transmute into the graphics characters of the Commodore fonts, as the big box above demonstrates. Remember, though,

```
240   if i=0 then quit
245   poke -6016, i : read i
250   poke -6015, i
255 endloop
260 endproc
265
270 proc lines_in
275 data 4,32,5,3,7,29,9,9,0
280 loop
285   read i
290   if i = 0 then quit
295   poke -6016, i : read i
300   poke -6015, i
305 endloop
310 restore
315 endproc
320
325 proc delay
330   x=cursor(1761)
335   input "PRESS RETURN TO GO ON",o$
340 endproc
```

that your printer has no idea of what's going on, and will output ASCII characters, not graphics.

The procedures at left remove spaces between text lines, put there to make text readable. You may remove them and put them back (see p. 69, <u>System Overview</u> manual). Some languages won't let you use integers above 32767 or hex, so we converted to negative decimal values at left. Here's the table showing how we got there:

Hex	Decimal	Neg.Decimal		Location Minus 64K bytes (Decimal)
E880	59520	-6016	=	(59520-65536)
E881	59521	-6015	=	(59521-65536)

We urge you to do these POKEs in subroutines or procedures which you can call from immediate or debugger mode--for when you STOP a program and are stranded without spaces it's far easier to call a routine than to type in the pokes--and get new eyeballs.

TELECOMMUNICATION FROM THE LANGUAGES Steve Zeller devoted his column, issue 11 (p. 162 ff), to telecommunications within APL, but found the process too slow for his purposes. He cried for help from a dedicated 6809 assembly language programmer. Help has arrived; the dedicated programmer is Loch Rose of 102 Fresh Pond Parkway, Cambridge, MA 02138, who has written APLCOM, an assembly language program which allows APL to pass data to and from the ML module. Loch then wrote BASICOM, which does the same for microBASIC. After Steve waxed enthusiastic over APLCOM and ye ed had tried BASICOM, we suggested to Loch that he write a universal program which would work in all languages, so he wouldn't have to develop a different version for the rest. As we go to press, Loch reports he has the first version up and running.

Why would you want to TC from language? (1) All the repetitive chaff--phone numbers, logon sequences with a mini or mainframe, your account numbers in a pay-my-bills scheme, access codes and such--can be put into program so that you need no longer deal with this mass of trivia; (2) all the stuff you want to upload and download can be identified and defined before you make a connection; (3) you can bring data, if you care to, right into the program which will process it. Don't get the idea you are going to stay on line whilst processing; that would be too expensive. Instead, think of the time and effort saved by defining what will do before you do it and by not having to refer to those lists of trivia you used to thumbtack all over the walls. These programs don't replace terminal programs such as NEWTERM, COM-MASTER and PETCOM, in which you can interactively chat and browse (as well as transfer files). They shine at repetitive jobs.

The ML modules to do the job are finished; you needn't concern yourselves with assembly language. You must, however, write a program in language to specify what's to be done. Loch has provided a demo program in mBASIC which shows, in a very simple way, how to use the capability. APLCOM is similarly set up. We hope the effort on UNI-COM is successful. The disk with APLCOM and BASICOM on it will be available August 1st. When and as and if UNICOM is worked out, it will be on the same disk. For copies, send $10 U.S. to Editor, PO Box 411, Hatteras, N.C. 27943. <u>State disk format: 4040 or 8050!</u> We'll report on UNI-COM next issue. The instructions for all ...COM programs will be on the disk.

THUD, CLUNK, CRASH DEPT. Some folks seem to think that if the wings come off SuperPET, the computer and drives turn into a smoking ruin, and you have to buy new ones. We keep getting letters saying, "I tried so-and-so and crashed; SuperPET seems to be okay, but I won't try THAT again..." If any harm could come to SPET from a program crash, ours would be a junkpile. Crash as often as you like; only your ego is dented. Flip to 6502 mode and back; RESET to program, and go. You don't learn unless you clunk, thud, crash at least once every session. We encourage Immelmann turns at altitudes of one foot with this column, in which we hope users will tell us how they knocked out their eyeteeth--to save others from similar fates.

Ever spend one whole day trying to get an assembly-language program to SYS from the languages, only to find that while it works fine, you always crash when you leave the ML module? Clunk, thud. We did. We always use JSR INITSTD_ to start an assembly-language program, which works fine from menu or in the monitor. But do not use it in a program which you SYS from language! The language itself initializes IO, and you needn't. We think (not sure) INITSTD_ resets the user stack pointer and stack, which handles bank-switched calls (in language, there are a bunch). Anyway, as soon as we took INITSTD_ out, our SYS modules ran fine and values on the U stack pointer returned from the vale of tears.

A GOOD THUD Butterflies, full moons, and io_status are ephemera--which means they don't last very long. In the loop at the left, if you're getting a string from a disk file, you'll find yourself staring at a blank screen forever after the last line$ comes off disk. The system variable io_status will give you a quick end-of-file signal when you try to 'linput line$' at end-of-file. Ignore it and it evaporates--for io_status is a <u>current</u> report. You must catch the report when it says "end-of-file", for io_status will change to report on the <u>next</u> operation. The instruction to QUIT must immediately follow the linput statement, else thou art in an infinite loop....

```
on eof ignore
loop
  linput #40, line$
  print line$
  if io_status<>0 then quit
endloop
```

SUPERCLUNK Six of you poor devils have ISPUG utility disks with a @*! dumbjohn on it. We wanted to give users a way to quit ALPHASYS when they got trapped in the program and could <u>not</u> remember the name of the file to be sorted. So we put a 'quit' on 'q' in, and told the program to JUMP to FINI, a label that wound up RTS'ing (we thought back to language) when it got that 'q'. We did it in a subroutine that had its <u>own</u> RTS.... Yeah. The address on the stack for the RTS was that to return to the ML program <u>from the subroutine</u>, not the final RTS back to language. If you six don't starve to death waiting to get out of that loop, send back the disk and we'll put on an ALPHASYS that really will quit. Really. Moral for dumbheads: if you want to quit an ML program in a subroutine, set a flag in the subroutine and jump to the final RTS from the main program. Yeah, we caught it on most of the disks. Blush.

HOW NOT TO MURDER THE EDITOR We had this great program which ran in the monitor in mED everywhere, but we always crashed when we left the monitor and tried to re-enter mED. Of course, we used all of user memory, from $0a00 on up...and it finally dawned that we'd wiped out those two low pointers in mED which tell the poor thing it has an empty file. As soon as we changed the ML routine so it loaded the four bytes starting at $0a00 with: 02 01 02 01, we no longer crashed the mED, and returned to it from the monitor happily. As Gary Ratliff noted in issue 12, starting on p. 188, those two pointers tell mED where its file doth start and end. If you wipe 'em out, you murder the editor.

DECREMENTING A LABEL?? We won't tell on the poor fella who sent this one in, but he set up a label with LABEL EQU *, and then did a DEC LABEL, and wondered why 1) he couldn't find label on his variable list, and 2) why he kept crashing. Poor thing wasn't decrementing a label at all, but the value at the address defined by LABEL, which was (of course) part of his program. Then he tried LABEL EQU 40 (since 40 was the number he wanted to decrement) and that didn't work, either, since it decrements whatever is at address decimal 40 down in the zero page. He finally got going by setting up LABEL FCB 40, which forms a single byte holding the value of 40, which he can decrement safely. Old hands may think this is funny, but beginners surely won't.

OOPS, AHEM and SMALL DISASTER DEPT. In issue 13 we published on pp. 217-219 a safe-bank-switch patch based on that sent in by Terry Peterson. Shortly after, we got a note from Joe Bostic: "There is a subtle yet potentially disastrous error in the patch on p. 219. The patch will indeed make interrupt routines in banked memory safe to operate but it will cause any other routine in bank-switched memory to fail if these routines need the interrupt disabled when they are called.

"The patch first disables the interrupt (SEI), which is good, but at the end of the patch it enables the interrupt (CLI) even if the interrupt was originally disabled, which is bad. The idea is to restore the interrupt flag to its original value. This technique can be used on any 6809 routine that needs an interrupt disabled but does not want to change the interrupt flag when it is finished."

```
safe_bank_switch equ *
 pshs    cc       ; Save copy, interrupt flag
 sei              ; Disable regular interrupt
 stb     $0220    ; Change 'bank in progress'
 stb     $effc    ; Change actual bank used
 puls    cc       ; RESTORE INTERRUPT FLAG!!!
 rts
```

Joe is absolutely right. We should have caught that one--and didn't. Terry comments that while the problem noted won't happen very often, it will happen. Use the patch above. Ah, well; another uuunk for the boobook.

A LENGTH PROBLEM IN RELATIVE FILES Jerry W. Carroll of 21735 Ybarra Road, Woodland Hills, CA 91346, notes a problem when you amend a record taken from a relative file and put it back into the disk file. Suppose we have a fixed file of 10 bytes (very small, but it saves space here). If we try to store ten periods in record 1, we find that the 10th is shoved into record 2, as shown at left.

```
1234567890  Byte count
.........   Record 1
.           Record 2
```

Why? Jerry and ye ed puzzled over this, and concluded that you must leave room in the disk file for the 10th character, which is a CR. The interpreter in microBASIC shoves a CR in at position 10 whether you like it or not, and then prints the 10th character in record 2. If record 2 happens to have a record in it, kiss it goodbye, for the 11th character again is a CR, which wipes out record 2. You'll not see this if you send your records to disk in numerical order, since Record 2, when written, overwrites the "lapped-over" character from Record 1. But you will see it if you write Record 2 first, and then write 1.

With that background, ask what happens if you linput a short a$ ('John') from record 2, change a$ to 'Joan', and refile a$ to record 2. Will it work? Sorry.

Though 'John' is only four bytes long, the record from disk with 'John' in it will be the fixed length of ten bytes. If you then replace 'John' by writing, in program, "a$(1:4)='Joan', and print the revised a$ to record 2, you erase record 3! Why? Gee, you added a CR to a ten-byte string. If you follow the example in the first paragraph, you'll see why. The solution sent by Jerry: truncate a$ to one byte less than the record length, and refile a$(1:9). You can be shrewd and clever and make 'Joan' into a$(1:5)--which also works.

LINPUT and INPUT bring in two different strings from any record. Assume we have no commas in the strings above. LINPUT brings in 'John', five spaces, and a CR, for a total length of 10 characters. INPUT 'John', and string length is 4. If we should make a$ equal to 35 X's and LINPUT it, the string length will be 35, but only 34 X's will be in that record plus the CR.

In short, LINPUT brings in the string, any spaces, and the CR at end-of-record. INPUT brings in only the string--minus terminal spaces and minus any CR. Bear the difference in mind when you form and revise REL records. The program below, taken from one of Jerry's, will demonstrate all this if you modify and run it. To illustrate what we said in issue 9, p. 134 about using TERMINAL to test REL files, this one is set up so you can see what happens in the file on screen. If you'd rather have the file on disk, change line 130. You'll have to hit RETURN on the INPUT or LINPUT lines of the program if you use the terminal as the file.

```
100 ! reldemo:bd. A demo of a string length problem in relative files.
110 !               PRINT STRING INTO RECORDS 1 THROUGH 3
130 print chr$(12) : open #2, "(f:35)terminal", inout
140 for i=1 to 3
150   print #2, rec =i, "John"     ! Try "if i<>2" and string "rpt$('.',35)"
160 next i
170 call printt ("First Pass,  ") ! Print the file to screen as a REL file.
180 !                       MODIFY RECORD 2
190 linput #2, rec =2, a$
200 a$(1:4) = "Joan"              ! Substitute a$(1:5) and see what happens.
210 print #2, rec =2, a$(1:34)    ! Remove the (1:34) and use plain a$.
220 call printt ("Second Pass, ") ! You must hit RETURN to input the lines
230 close #2                      ! from the Terminal screen.
240
250 proc printt(file$)
260   for i=1 to 3
270     linput #2, rec =i, a$
280     x=cursor(801+n) : print file$; "Record";i;"= ";a$;" length =";len(a$)
285     n=n+80
290   next i
300 endproc
```

A COmmon Business Oriented Language
by Michael R. Cook

2045 Cambridge Drive B-21
Lexington, Kentucky 40504
(606) 252 0917

[Ed. Mike is a professional programmer, and an old hand at COBOL. We asked him to send a short introduction to the language and to stand by for questions. He says the Waterloo manual on COBOL is so complete you can learn the language with no other references. Considering its thickness, we're glad to hear it!]

COBOL was created primarily for three reasons; 1) Portability, 2) Maintenance, and 3) Productivity. The development group consisted of computer manufacturers, major business users and the U.S. Dept. of Defense (the primary user of the finished product). Portability would benefit all users as conversion costs would be minimized, and in this COBOL has been reasonably successful. The expectations of productivity and ease of maintenance were naive, however. It is well known that data processing shops across the country are looking for ways to increase productivity and that maintenance of programs consumes over 70% of a typical shop's programming resources. In recent years, the benefits of structured programing (as taught by Paul Noll or Edward Yourdon) in increasing productivity have been proven.

Self-documenting code is considered by many as COBOL's main strength, but the fact is (as in any language) that this depends on the programmer's abilities. In hands of experience, COBOL can be highly readable and understandable, while the novice may produce code that rivals APL's cryptic nature. The primary weakness of the language is not its verbosity but that it allows an obscure style of coding (e.g.; GO TO, ALTER) to be used, unlike PASCAL, which doesn't support such statements. Experience also helps you to eliminate unnecessary coding that only confuses the subject (e.g.; '=', not 'is equal to') and create meaningful data names ('ws-line-counter', not 'ln-ctr'). The mCOBOL manual does an excellent job of introducing the novice to a clear coding style, so follow it!

File processing and reporting are COBOL's true calling and its primary use (unless you are programming real-time applications) and so most ISPUGers may well wonder of what use the mCOBOL interpreter will be. Suggestion: Let mBASIC handle the user interface in your applications (its true calling) and write some reports and file maintenance programs in mCOBOL, and learn a new language! Using mCOBOL will give you a better idea of the differences between languages, their strong and weak points (all have them!) and knowledge of a language that still commands a strong position in the job market.

COBOL is celebrating its 25th anniversary and is the D.P. industry's most common language and favorite whipping boy. I suspect that until the true fourth generation languages appear (and radically change D.P. in the process), COBOL will continue in its role as the Rodney Dangerfield of languages (no respect!), keeping the tapes and disks spinning and the financial reports on the desk of the director of finance.

I hope that this has given you an interest in exploring the language and not the desire to scratch the COBOL PRG file from your language disk! I welcome any suggestions that you may have for future articles.

ANATOMY OF MICROBASIC PART 2

by Gary Ratliff, Sr.
215 Pemberton Drive
Pearl, Mississippi 39208

[Ed. Gary wrote the original ANATOMY in one part, far too long for one issue. To help those who may not remember details of Part 1, we print below a commented mBASIC program which shows program lines and what the VARIABLE table holds, before and after a RUN. Most of the points covered in Part 1 are so summarized. Note the location and nature of the VARIABLE table, which resides a few bytes above the top of the program itself.

[In the program below, find a few abbreviations: **LoL** indicates Length of Line in bytes; **Bptr** means Back Pointer in bytes to the start of line. The simple program at left generates the code shown below. mBASIC lines start at $0a01. Let Gary take it from here.]

```
10 a%=10
20 b%=20
```

30 a%=a%*b%   Let me point out the similarity between mBASIC lines and the method of text compression for the mED. The chain of forward and backward pointers is similar but for the addition of a line number in mBASIC which is not used in the mED. The lowest (and illegal) line number of 0000 shows the start of program, while the highest and equally illegal FFFF shows the end of program lines. A null program thus is: 04 00 00 03 04 ff ff 03, much like a null workspace in the mED, which is 02 01 02 01. The leading byte (02 for mED) is the byte count or length of line (LoL), which for a null mBASIC program must become 04 for the two extra bytes in 04 00 00 03. First, the program before RUN.

```
Address:      START of BASIC Lines
           LoL                        Bptr ¦ LoL     Line    10    Real Var. (continued)
;0a01      04      00      00      03 ¦  09      00      0a      00
--------------------------------------------------------------------------------------
a at Loc. 01      =   numeric    10      Bptr ¦ LoL     Line    20    (continued)
;0a09      01      95      8d      0a      08 ¦  09      00      14
--------------------------------------------------------------------------------------
      Var (b)  at 05     =    numeric     20     Bptr ¦ LoL     Line (continued)
;0a11      00      05      95      8d      14      08 ¦  0c      00
--------------------------------------------------------------------------------------
           30    Var. at 01 (a)  =   var. a at 01   times   Var. (continued)
;0a19      1e      00      01      95      00      01      96      00
--------------------------------------------------------------------------------------
b loc. at 05     Bptr ¦    End of BASIC lines    ¦ LoL  A separator/pointer,
;0a21      05      0b ¦ 04      ff      ff      03 ¦ 04      00    (continued)
--------------------------------------------------------------------------------------
not defined, fitted   ¦            START of Variable Table
between program and   ¦ Integer a, length of 1.   ¦ Integer b, length of 1,
variable table. Bptr  ¦ 2 bytes reserved for val. ¦            (continued)
;0a29      00      03 ¦ 41      61      00      00 ¦ 41      62
--------------------------------------------------------------------------------------
2 bytes reserved for  ¦  End of Variable Table before RUN. Note that the values
dynamic values.       ¦  of Variables 'a' and 'b' do not appear in the Variable
;0a31      00      00 ¦  table. See below what happens when line 30 uses them.
======================================================================================
To right, the variable¦            START of Variable Table:
table after a RUN:    ¦ Integer Variable a, length 1, now has
                      ¦ value of $c8, or dec 200 ¦ Int. var. b, length 1, has a
;0a29      00      03 ¦ 41      61      00      c8 ¦ 41      62
--------------------------------------------------------------------------------------
table val. of dec. 20 ¦  The vertical bars above "¦" indicate the end of an
;0a31      00      14 ¦  mBASIC line or the end of a VARIABLE Table definition.
--------------------------------------------------------------------------------------
```

The chaining of the lines by byte count means that the forward chain may easily be traced to the end of mBASIC, where FFFF will always be found. This suggests a very simple program to find mBASIC end-of-program.

```
membeg_  equ $20    ; pointer to start of mBASIC forces direct addressing mode.

ldx      membeg_    ; load the pointer.
leax     1,x        ; if membeg_ holds 0a00, then mBASIC starts at $0a01.
loop                ;
  ldb    ,x         ; get length of line byte (the first will be 04).
  ldy   1,x          ; one beyond this is the line number value.
  cmpy  #$ffff       ; when we find this, we're at end of mBASIC lines.
  quif  eq
  abx                ; add length byte to location to find next length byte.
endloop
abx                  ; add final 4 bytes to skip past 04 ff ff 03 to next xx.
tfr     x,d          ; take end found and transfer to D register for math.
addd    #??          ; value indicated by ?? must be inserted to find values
                     ; wanted in variable table or elsewhere.
swi                  ; register dump; address of desired value is in D register.
                     ; Y will contain FFFF to verify end of program.
                     ; X will contain pointer to byte immediately following the
                     ; 03 backward pointer of line 04 ff ff 03.
```

Those who wish to pass parameters between ML modules and mBASIC programs must investigate what happens when arrays, floating point, strings and functions are used and learn where values or pointers to such items are to be found. To do this, remember the following table, which summarizes the way Waterloo identifies the variables and user-defined functions <u>in the 1st byte of the VARIABLE table</u>. Using this data, we'll generate some functions and give an example of how to locate their addresses.

Bit Number: 7	6	5	4 3 2 1 0	<u>Meaning:</u>
0	0	0	Bytes 0-4	Floating Point or Real Numeric
0	0	1	reserved for length	String
0	1	0	of variable name.	Integer Numeric
1	0	0	5 bytes can hold a	Real or Floating Point Function
1	0	1	a maximum character	String Function
1	1	0	count of 31.	Integer Function

You will find the values defined above in the VARIABLE tables below, after we record the small program at left, and then RUN it. Some additional +128 tokens in the program lines now become clear: $86 shows a numeric value, integer or real; $84 indicates a string value; $d1 identifies a function. We're now able to read almost every character in the program and in the variable table. Here is what we see before we RUN:

```
10 def fna=1
20 def fna%=1
30 def fna$="one"
```

```
                  Start of BASIC     ¦  LoL    line   10    d1 identifies function
;0a01     04      00      00     03  ¦  0c     00     0a     d1
-------------------------------------------------------------------------------
Reserved for ?    ?      at loc. 1      =    Numeric    1    Bptr ¦
;0a09     00      00      00     01     95     8d     01     0b   ¦
-------------------------------------------------------------------------------
          LoL     Line    20  function                at loc. 5 in Variable Table
;0a11     0c      00      14     d1     00     00     00     05
-------------------------------------------------------------------------------
          =       Numeric  1     Bptr ¦ LoL    line   30     function
;0a19     95      8d      01     0b   ¦ 0f     00     1e     d1
-------------------------------------------------------------------------------
           ?       ?      at loc. 9     =    string, len 3  "o"
;0a21     00      00      00     09     95     84     03     6f
-------------------------------------------------------------------------------
          "n"     "e"     Bptr ¦   End of BASIC lines     ¦  start of separator
;0a29     6e      65      0e   ¦  04     ff     ff     03 ¦  04
```

```
-------------------------------------------------------------------------------
                                 |           START of Variable Table
        end of separator         | Numeric fn is 100, len 0001| Integer fn 110 len 001
                                 | Binary 1000 0001 is 81.    | Binary 1100 0001 = $c1
                                 | Real Fn   a     reserved   | Int. Function
;0a31     00      00      03     |  81      61     00     00  |  c1
-------------------------------------------------------------------------------
                  reserved        |  String fn is 101, len is 0001
    named a       bytes           |  Binary 1010 0001 is $a1. |
                                  |Str Fn    a     reserved   |  12 bytes in Variable
;0a39     61      00      00      |  a1      61     00     00 |  0c               Table
===============================================================================
```

Here is the Variable Table after a RUN. It contains the ADDRESSES at which the functions will be found, apparently so they may be executed when called.

```
-------------------------------------------------------------------------------
   End of Separator               |           Variable Table (Note system addresses)
                                  |Num Fn   a at address 0a      05  |  Integer Fn
;0a31     00      00      03      | 81      61            0a     05  |  c1
-------------------------------------------------------------------------------
          a  at   0a      11      | Str fn   a at  0a      1d |  12 bytes in Variable
;0a39     61      0a      11      |  a1      61    0a      1d |  0c            Table
-------------------------------------------------------------------------------
```

We now have a handle on some tokens, the way lines are stored, the location and contents of the variable table, know where values or addresses can be found, and are aware that values of variables appear both statically in lines and dynamically in the variable table. We can identify both the types and lengths of all the different types of variables, and have a program means of getting to the variable table to find either the values or the addresses we want--at least in part. Next issue, we'll look at dimensioned variables.

INPUT FROM WATERLOO
Comments and Clarifications

Too late for issue 13, we received a long letter of comment on material in issue 12 from WATCOM Systems. We print a summary below, referenced to the articles in issue 12. **Attend with care to the notes on SYS from language!**

USE of RND in mFORTRAN In I, 184, Stan Brockman noted the pattern in RANDOM numbers generated in mFORTRAN. Waterloo comments, "The argument to RND is a seed which is used to generate a pseudo random number. The seed value is not updated whenever the function is called. To generate a sequence of pseudo random numbers successive invocations of RND should involve the previously generated number. If you pass the same value to RND on every invocation, the sequence will not be random. In fact, as you guessed, the only element of randomness is the time-of-day clock. This won't be very random if the function is invoked at regularly spaced intervals. The value returned by RND should be used as the next argument

```
RANDOM=SEEDVALUE
LOOP
  RANDOM=RND(RANDOM)
  ...
```

to random. In fact, as you guessed, the only element of randomness is the time-of-day clock. This won't be very random if the function is invoked at regularly spaced intervals. The value returned by RND should be used as the next argument to RND. The following (see left. Ed.) is an example of the correct way to use RND.

GET # FROM TERMINAL in mBASIC In I, 185, we noted the unusual behavior of a GET from terminal in mBASIC, which is highly dissimilar to a GET from keyboard or a plain GET in mBASIC. A GET # from terminal will accept up to 80 characters before RETURN is pressed, and will (1) return the ordinals and (2) print the 'gotten' characters to screen. Waterloo comments: "The terminal device is a buffered device. MicroBASIC requests the operating system to read a record from the terminal when GET is encountered the first time (mBASIC calls the system routine GETREC_). The operating system does not return a record until the user presses the RETURN key. During this time, mBASIC is blocked from executing. The user can ...move about the screen and edit any line.... It is only when the RETURN key is pressed that mBASIC resumes execution. It now has a record of up to 80 characters. The GET # statement returns the first character in this record. Subsequent executions of GET # cause the second, third, etc. characters...to be returned. This continues until all the characters in the record have been returned....

"The keyboard device, on the other hand, is not buffered. This is the device that a microBASIC statement such as GET A uses.... The same effect as GET A can be achieved if the OPEN statement refers to the device "keyboard" instead of "terminal".... You can think of the terminal as a record-oriented device and the keyboard as a character-oriented device."

LOADING MACHINE-LANGUAGE SUBPROGRAMS In issue 12, p. 185, we showed two ways to load ML modules for use in the languages--one from main menu; the second from the monitor in language. The program printed indeed worked okay that way--BUT we now know there's a booby trap a-waiting if you try it in the monitor. Waterloo comments that the monitor alternative (loading the ML module after the language is loaded) "will not succeed...for any interpreter except possibly microBASIC.

"To understand why, let us describe how memory is set up by the interpreters which use the microEDITOR as the primary means of entering source text. As soon as the interpreter is loaded into memory, it does some initialization. In particular, it allocates 4 buffers at the high end of RAM (see below)...If you slip

Pointers/Buffers:		Typical Address:	Buffer Bytes: (added by Ed.)
HighBound =$A0	Highest address for source code/symbol table.	$7E6E	
N_Buffer =$A2	Filename buffer.	$7E6F	40
S_Buffer =$A4	Command save buffer.	$7E97	120
Sub_Buffer=$A6	Substitution buffer.	$7F0F	120
Re_Buffer =$A8	Regular expression buffer.	$7F87	120

into the Monitor after loading an interpreter and dump the locations...you'll see the five memory pointers...If you load a machine language program anywhere in this range then you would be lucky if it survived during normal editing. When a program is run, these memory locations (shown above) are used for other purposes. When return is made to the microEDITOR, after execution, these buffers are reallocated."

We got away with monitor loading of the modules printed in issue 12 because we did no editing. So be warned: in mPASCAL and in mFORTRAN, the methods following are the only safe ways to load an ML module to SYS from those languages:

1. Load the ML module from main menu or from monitor at main menu; then load the language. (Here, we assume you or the module reset MEMEND_ to protect the module.) All necessary pointers are set when the language is loaded.

2. Alternatively, if the language has been loaded, leave it with a "bye", and load the module from menu (monitor included); then use program RESET to

return to language. (Again, we assume a reset of MEMEND_ at menu).

We tested exhaustively, and found both methods above are reliable. Nothing else is--including a RUN in the language--which seems to reset the pointers Waterloo mentions, but inevitably allows the ML module to be overwritten.

```
*stringtest:f
character c

c=rpt('*',9000)
end
```

The comments above do not apply to microBASIC. The method shown for loading ML modules in issue 13, page 219 ff., is reliable. Those who wish to explore the problem are advised to run the little mFORTRAN test program at left after they reset MEMEND_. It forms a giant string of 18,000 asterisks (yes, 18,000--not 9,000) which inevitably overwrites memory above MEMEND_ as reset unless you use one of the two methods above to force the operating system to allocate and protect the memory space above MEMEND_ in mFORTRAN and in mPASCAL.

DETERMINING THE START OF AN ML MODULE Both Waterloo and Loch Rose commented on ye ed's use of the LIST (.lst) file to determine where the language portion of an ML module starts (see p. 186, issue 12). Waterloo says "There is a less awkward way to determine a starting address. The programmer simply uses the XDEF statement. After linking a program, just peruse the ".exp" file for the label(s) in question (see example, left). The exports file contains all externally defined symbols and their addresses. This would include "ml" in the above example." We didn't do it that way for two excellent reasons: (1) We've had a lot of questions on "what the heck is the .lst file for and what is all that stuff in the file?" and wanted to show specifically how to use it, and, (2) we often forget to XDEF and need addresses. If you don't know how to use the .lst file, you'll never get them. The XDEF statement is handy once you've debugged your code and know what addresses you want to define and later use.

```
;previous code...
xdef ml
ml   equ *
;following code...
;........
```

HEY, GARY, ARE YOU AWAKE?? Did you notice last year when plain Gary Ratliff suddenly became Gary Ratliff, Sr.? Well, he has become Gary Ratliff, Sr.^{2}; our coding expert has successfully transmitted some DNA code, and had little sleep after the dash to the maternity ward (five wrong turns). It's a boy. Anent his article on text compression in the microEDITOR (Issue 12, p. 188), Waterloo says "It may surprise you to learn that the Editor, in fact, knows absolutely nothing about source program management (i.e., encoding of keywords, variable names, constants, etc.). The editor and interpreter are bundled together in a cooperative manner...the interpreter handles the allocation of memory and encoding for source lines. The EDITOR requests that the interpreter decode and place in a buffer ($400) each source line that it wishes to examine. The EDITOR expects a string composed of 0 or more characters to be placed in this buffer. The string is terminated with a NULL ($00)...It does not matter to the EDITOR how a line is represented in the workspace or in a file...it deals with...the decoded string placed in the buffer.... It is this strategy that allows us to use the exact same editor in the four language interpreters. Each interpreter maintains its own scheme for encoding of the source program [emphasis added] and the EDITOR need not be aware of what it is." Okay, Gary. After you change the diapers, you can go back to sleep.

AN OVERVIEW OF FORMAT CONTROL IN FORTRAN

By Stan Brockman
11715 West 33rd Place
Wheat Ridge, Colorado 80033

[Ed. Many a letter has arrived on format control in mFORTRAN, one of the matters known best to the Cabots, Lodges, and God. We asked old FORTRAN hand Stan Brockman for a summary and overview on the matter, which knits the fragments into a whole.]

There are two modes of Input/Output (I/O) when using microFORTRAN (mFOR) on the SuperPET: (1) data format under your control and (2) default format under control of the interpreter [the last is called List-Directed (LD)]. The mode in which you control format I'll call FORMATTED I/O; the default mode, LD format. Formatted I/O is a particularly powerful feature of FORTRAN. It lets us specify our own rules when we read, write, or print data; we needn't accept the defaults built into mFORTRAN. Page 151 of the manual covers FORMAT statements tersely.

List-Directed Input and Output

LD READ statements specify default format and take one of two general forms: 'read,x,...' or 'read*,x,...'. (The asterisk means 'default format', or 'for heaven's sake, don't use any FORMAT statements on this READ!'). The data type of the variable (character, integer, real) determines how the data are stored in the mFOR default format for that data type. You find the data type, of course, at the start of program or subroutine, where you list variables by their type--hence the phrase, "List Directed". See manual, p. 143.

These default, LD READs assume that data will be in fields delimited by commas or by a CR at end-line. Let's look now at the default format which LD gives us.

Character variables are assigned all printable characters which you enter from the keyboard, but minus any leading blanks or spaces. In short, they are left-justified, and padded to the right with blanks--if any. In this article, I use the carat "^" to show blanks or spaces in variables and in records, as at left. If you LD print two character records (c="judge", d="meant"), with print, c, d, you create a single mis-spelled word, "judgemeant", which you may separate with right-padded spaces or by concatenating with a space. If you READ several character variables at one time, separated by commas, any blanks preceding commas become part of a variable, but blanks after the last entry do not; if we READ,a,b,c and entries are: ^NOW^,^THEN^,^NEVER^<CR>, the character variables stored are: NOW^, THEN^, and NEVER.

```
Your char. input is:
"^^^What prints?^"

Record is stored as:
What prints?^
```

Numeric variables LD handles differently. It trims both leading and trailing blanks from integer or real variables, however many spaces you may leave between the commas in a reply to a read statement. Suppose we are asked to enter an integer "i" and a real "a" as at left, and respond with the two values shown. Despite the blanks we leave, the records will contain none. But--saints preserve us--integers and reals LD print on output in two entirely different ways.

```
To READ, i, a
we respond with:
^12^,^144.346
```

Integers always print in a 7-space zone. Since 32767 is the maximum integer, six spaces are enough for value plus sign; the additional space serves to separate the records, however we print them. At left, I illustrate with integer val-

```
(index to print zones)
1234567123456712345 67

^^^^^^2^^12345^^^^300
^^12345
^^^^300
```

ues. The index line shows you the 7-space print zone; we print three integers on the first line at the left with "print i, j, k". If, instead, we print each as a separate line, we get three rows, in which the tens, hundreds, thousands, etc. are aligned by column. Such a format is easy to read and understand. If only the LD format for reals were compatible!

Real values are shown in a different LD format. We're limited to a maximum of eight significant digits, a sign and decimal; one space separates all values. We thus find an 11-space print zone when we LD print reals, whether we LD print them in columns or in rows. The decimal points aren't aligned, and any integer we may print looks forlorn and out of place (see the final '12', at left). Last, we find reals of more than eight digits are both truncated and rounded. A value of 12.3456789 is seen as 12.345679. In sum, LD format is simple but often isn't easy to read--especially when scientific format prints in a 16-space zone with eight significant digits.

```
^^.12345678
^^12.345678
^^12345.678
^^^^^12*
(* integer)
```

Formatted Input

To format input you must describe 1) where and how big the data fields are to be, and 2) the types of data in each field of the record. Your description is a format specification, which is made up of a sequence of edit descriptors, or EDs. Before we describe the EDs, remember the following:

Numeric Data: Embedded and trailing blanks are translated to zeros. Leading blanks are ignored, as they are in LD input. It's good practice to right-justify numeric data so that blanks aren't converted to unwanted zeros. Decimal points aren't permitted in data sent to integers; if they're present, you get an "Input Conversion Error." In contrast, LD input allows, but truncates at, decimals. All decimal points are included in the count of a formatted field.

The main difference between LD and Formatted input is that a format statement--instead of commas--subdivides the input record; it does this with edit descriptors (EDs) which define and describe the fields. Commas, if present, are read as commas in alpha (character) fields, but cause errors if read in numeric fields. I list the EDs at left. The 'A' descriptor reads characters and stores them in the same form as the input record. 'I' describes a numeric field whose characters are to be translated and stored as integer data. 'F' describes real values; 'E' formats reals on output to scientific format. 'X' you use to ignore characters. See manual, pp 152-156, for a fairly complete but terse description of EDs, unfortunately without many examples.

ED:	Used for:
A	Alphabetic data (character)
I	Integer data
F	Real data
E	Real data, Scientific Format
X	Ignore characters

FORMAT statements can appear anywhere in a program after you have declared your variable types; you may use those statements throughout your program. The EDs above, when strung together, form a format specification, which can take three different forms--two closely related. The first form has been used the longest and the most often; we now examine it. The READ line at left says to read a record from the keyboard, to use format statement #10 to translate

First Form of Format:

```
   read 10, d,k,y
```

```
10 format(a8,1x,i5,1x,f6.3)
*index: 12345678
   d = "^TEST^1^"    *char
   k =  780          *int
   y =  12.345       *real
```

the record, and finally to assign the resulting values as formatted to d, k and y. Assume that the record to be read is "^TEST^1^,^^78^,^12345". Remember that the carats represent blanks. I've added an 'index' line to show the character count. FORMAT 10 is the road map; let's take it ED by ED. "a8" says "assign the first eight characters to the character variable 'd' and store them as characters"; "1x" says "ignore one character" (the comma); "i5" says "translate the next five characters into an integer, store it as such, and assign it to the integer variable 'k'"; "1x" ignores another comma; "f6.3" says to read six values, assume a decimal point three digits left of the last value, and to translate and store the values as real data assigned to variable 'y'.

Now consider what is assigned to the variables. The character variable "d" is equal to the input "^TEST^1^" because--unlike an LD read--the leading blank is not discarded. Integer "k" is zero-filled on the right while its leading blank is stripped, hence "78^" becomes "780". Real "y" is given the value 12.345 when read (not the value of 12345.0 as if LD read) despite the absence of an actual decimal point--because the ED "f6.3" commands an implied decimal point three places to the left of the rightmost, least significant digit in the entry--if and only if there's no decimal point in the entry.

On input, both the "E" and "F" EDs operate in the same way: both describe the width of the field to be read and both emplace a decimal point if none is present in the field read. Yet, if a decimal is present, it takes precedence over that implied by the ED. If either "f5.4" or "e5.4" is used to read a field of "12345", the result is the same: "1.2345" is stored and assigned to the associated variable. The same EDs, if used to read the field "123.4", will store and assign the value "123.4".

Now, a word of caution. The actual length of the last field of an input record should be equal to or greater than the field width specified in the ED. MFOR inflexibly insists that it receive all the characters it was told to expect, and if it doesn't get them after a couple of tries, it will pout, sulk, and then announce an error, "Specified field width is too big," and quit. With that warning I end the discussion of the first form of FORMAT statements.

The second form of format specification uses character variables to contain the format EDs. You must define the character variable with an appropriate string of EDs. You may then use it with any input or output statement elsewhere in the program. Note the character variable "fmt1" in the demo program below and its assigned ED values. They're identical to the format spec we use in #10, above. We can use "fmt1" as shown below. This is the second form of FORMAT statement.

```
Definition of character var.
fmt1 = '(a8,1x,i5,1x,f6.3)'

Second Form of Format:
read(5,fmt1) d,k,y
read fmt1, d,k,y

Third Form of Format:
read(5,'(a8,1x,i5,1x,f6.3)') d,k,y
read '(a8,1x,i5,1x,f6.3)', d,k,y
```

It performs exactly as does format #10, and in "Second Form", left, reads from the terminal (default unit 5) or from the keyboard. At this point, examples 23-26 in the manual should be easier to understand. You can use the manual examples (Third Form) if you wish but the third form is clumsy and cannot be re-used elsewhere in your program, while the first two forms can be employed as you wish.

You find another advantage when you handle format with a character variable; you can substitute one such variable for another as format changes. The character variable "how", for example, may in one place equate to "fmt1", in another to "fmt2", and in a third to "fmt3", etc. READ, WRITE, and PRINT statements employing "how" may thus handle a number of different formats.

<u>Odds and Ends in Output</u>

Next, we look at some of the seemingly mad things which occur on output in mFOR, and at a few output odds and ends we haven't discussed.

For starters, let's output the character variable achar = '12345' to the screen. If you say 'print, achar' or 'print*,achar' or 'write(6,*) achar', five characters appear on screen. But--if you say "print 'a5', achar" or print 'a', achar", you clear the screen and print only '2345'! What goes on??? What happens to the 1? Why does the screen clear? Well, hey! you've just run into a Carriage Control Character (CCC)! That missing '1' is a CCC to clear screen and home the cursor, and we printed it on column 1, the only column of the screen where CCC's have any effect. Hurry, read the manual, p. 147. You'll find <u>five</u> leading characters which control screen output in column 1. You can avoid unwanted output surprises by printing a space " " at the beginning of a formatted output; the space says "print this record where the cursor is." If we say: print "' ',a5", achar, we no longer lose the character 1 and we don't clear the screen. Each output line or record needs its own CCC (remember that a blank--" "--is a CCC).

You should use the Carriage Control Characters (CCCs) only with formatted output, where you can control the first character of a record. CCCs work only with the SPET screen, for most printers don't respond and simply print them as normal characters. Now that you know what the first character can do, you'll understand why all LD output prints a <u>blank or space</u> as the first character of any output record. If LD didn't do this, every 1, 0, + or - you send could muck up output. LD format automatically insures that you get single-spaced screen output.

Don't overlook the slashbar '/'. When inserted in a format statement, it causes a CR on output and substitutes for a comma where used in the format statement.

You can be trapped by the width of the output field demanded on the "E" ED. That field must be <u>at least</u> the sum of 7 plus the number of digits which will appear after the decimal point in the field. Example: suppose the number "-123.45" is to be output in "E" format and that all five digits are to appear in the output. The width of the field must therefore be 12 (7+5); the ED must be "E12.5". You will print "-0.12345E+03".

A couple of final notes about formatted output: 1) The EDs may specify field sizes that are larger than required; you thus easily make sure that the space is large enough for an unexpectedly large value and you separate data values with "white space" (of course, the 'X' ED will do that, too). 2) You can specify the number of times a given ED (or a group of EDs within parens) is to be used with what the manual calls a "repetition factor," the effects of which the manual neglects to explain. Shown at left, below, is an example of a "repetition factor". It prints data values <u>on the same line or record</u> for the number of repetitions called for in the format spec. The "3" at left, in "3a7", is the repetition factor. If we take the "3" out, we find each "Char n" prints on a separate line or record, not in one row.

```
character b(3), fmt1
fmt1="(' ',3a7)"
do i=1,3
  b(i)="Char "//cnvi2c(i)
enddo
```

```
print fmt1, b
*printout shows below:
Char 1 Char 2 Char 3
```

You must use repetition factors with the 'x' ED. While 1x ignores one space and 3x ignores three spaces, x alone will not work.

There's a shorthand way to output character data. You may employ the "a" ED by itself, with no explicit statement of width of field. For character variables only, this "a" shorthand accepts an output field exactly as long as the string itself--no more, no less. It is an LD character output, of sorts. Under certain circumstances, "a" may be used as an ED on input, too, but the problems in doing so seem to me greater than the advantages, and I don't recommend it.

I've neglected the "T" formats (p. 156, manual), but I suspect that by this time you understand the fundamentals and can easily find out how to use them.

EVERYTHING YOU WANTED TO KNOW ABOUT THE SERIAL PORT... and were afraid to ask is found in a new WATCOM Technical Note, The SuperPET Serial Port, written by Jack Scheuler. We just got our copy, and are delighted. It's typeset, written clearly, and covers everything from the 6551 ACIA and its registers through the vagaries of wiring the RS232C for modems and printers and other computers. You will find an example of how to set and employ the ACIA and port in language, a demo of how to handle interrupts, and a set of assembly language routines you may use as building blocks to write a device driver or terminal emulator. Excellent reference work. $20 U.S. or Canadian. From WATCOM, 415 Phillip St., Waterloo, Ontario, Canada N2L 3X2.

TO BE OR NOT TO BE : REVISED LANGUAGES A NEW OPERATING SYSTEM? Those who previously got infoWAT or its sister publication WATNEWS, since replaced by one publication, WATCOM News, have seen the survey sent out by WATCOM Systems, which asks if you're interested in Version 2.0 of the various SuperPET languages, and how much you'd be willing to pay. So far, WATCOM has mailed about 1000 copies of the survey, and has received 90 replies--a near-fatal record of indifference, since WATCOM is a commercial enterprise which must pay its own way, having no income from the University and no subsidy from government.

Though we bought SuperPET with software bundled into the price, Commodore isn't involved in V2.0. WATCOM's payback must come from us. We've seen different figures on the cost to develop a V2.0 language, ranging from $50,000 on up. A total of 29 people have said they'd buy V2.0 mBASIC at $250 per copy. Gee; WATCOM's return would amount to $7250. Anyone for charity?

So, with this issue we make one last grab for V2.0. A survey form is attached. Fill it out and send it back to WATCOM. If enough of you respond, we might get one or more V2.0 languages. If not, well... Before you can respond, however, you should know what V2.0 will do that V1.1 software won't; see the summary in the pages which follow. Last, consider compilers. We understand that compilers for WATCOM BASIC and mPASCAL may soon be available on the IBM PC (versions of those which run on IBM VM/CMS and VAX/VMS); but clearly we won't see compilers on SPET until or unless V2.0 languages become available, because WATCOM won't attempt to support two different sets of compilers, one for V2.0, and another for V1.1.

Schools comprise the single largest group of SuperPET users; one with 40 SPETs can't afford to buy 40 copies of one V2.0 language, let alone 40 copies of all.

	First Column: First Year Fee		BASIC COBOL FORTRAN		Right Column: Fee in Subsequent Years	
Quantity	APL		Pascal		WATFILE	
10	1350	350	1100	300	1300	350
20	2150	550	1700	400	2000	500
30	2800	700	2200	600	2600	650
40	3450	900	2700	700	3150	800
50	4000	1000	3100	800	3650	900
100	6500	1600	4800	1200	5700	1400
200	10600	2600	7400	1800	8800	2200
300	14100	3600	9600	2400	11400	3000
400	17200	4400	11600	2800	13600	3600
500	20500	5000	13500	3500	16000	4000
1000	34000	9000	21000	5000	25000	6000
2000	58000	14000	30000	8000	38000	10000
3000	78000	21000	42000	12000	51000	12000

WATCOM offers a solution to IBM PC users (the data at left is representative but details may well differ for SuperPET). For the fees shown, you get 1) one copy of software and a license to run on a certain number of machines for one year, 2) updates of software, 3) WATCOM support by hotline or letter, 4) manual updates, and 5) WATCOM news. You get one manual, and can buy more. If you figure cost per machine per year, you'll find that it runs from $35 per PC per year with 10 machines to $16 per PC if you have 100 and use APL. The other languages cost less. It may cost more to scrape bubble gum off the chairs. If you need more information, write WATCOM Systems, Inc., 415 Phillip Street, Waterloo, Ontario, Canada N2L 3X2.

You have the essential facts in hand. Please fill out and mail the survey form.

* * *

OS-9 : A UNIX-TYPE OPERATING SYSTEM We recently had a call from Gerry Gold of Toronto, the head of the SuperPET Chapter in TPUG, about making the OS-9 operating system available for SuperPET. OS-9 was written by Microware Systems Corporation in collaboration with Motorola, to give the 6809 an operating system which takes full advantage of the chip's capabilities and instruction set. It never really caught on because manufacturers jumped to the 16- and 32-bit microprocessors which came along about the same time.

OS-9 nevertheless has a good reputation. MICRO magazine for June and July, 1983, carries an excellent two-part article on it by Stephen Childress; and another in November, 1983. While OS-9 seems flexible and powerful, and a fair amount of software is available (it runs on the Radio Shack CoCo), we don't know how much of it might run on SPET, how the software would be converted to Commodore disk format, or what hardware/software changes will be involved.

To implement OS-9 on SuperPET, our Toronto friends must 1) obtain permission and license from Microware, probably for a fee, and 2) write the code which adapts it to SuperPET. The effort will not be worthwhile unless the work can be sold to SuperPET users. Gerry guessed that a sale of at least 200 copies of OS-9, as adapted, would be necessary to defray costs. We won't say more until we have more facts.

* * *

SUMMARY OF V2.0 CAPABILITY We compress below the major features of the V2.0 languages, as they run on the IBM PC and certain mainframes. Understand that any V2.0 for SPET, if released, will be limited by SPET's architecture. **APL:** Random array file system similar to Sharp's. Quad FMT for report formatting from Sharp; extended string search/substitution; verification of input numerics with Quad VI

SuperPET Questionaire

Version 2.0 of the WATCOM languages, APL, BASIC, COBOL, FORTRAN and Pascal have been produced for other computer systems. We would like to get some feedback from SuperPET users on whether or not a SuperPET version of WATCOM languages is desired, and if so, which of the languages would be most useful in this context.

As part of our commitment to support SuperPET users of our languages, we would like to be able to offer new and exciting programming tools. However, it is important to know how many people would like which products and at what price. Recent products such as the 6502 Development System, PIP and the Terminal Emulator have had low response and we have not recovered the costs of development. That's why feedback is so important. We need to know what you want and how many of you want it.

By filling out this form, you will help us in determining the future of WATCOM product development for the SuperPET. Thank you.

1. Have you filled out and returned this form before? ______________________

2. Do you personally own one or more SuperPETs? How many? ________________

3. Do you currently employ one or more SuperPETs at an educational institution or other facility? How many? ______________________________

4. Do you use the SuperPET for business (____), education (____), or development (____) (check applicable ones)?

5. How often do you use the following WATCOM products?

	Often	Seldom	Never	Don't Have It
APL				V1.1
BASIC				V1.1
COBOL				V1.0
FORTRAN				V1.1
Pascal				V1.1
Editor				V1.1
6809 Development System				V1.1
6502 Development System				
PIP (disk/file utilities)				
Terminal Emulator				

6. Would you be willing to purchase version 2.0 of:

	Yes	No	Reason
APL (U.S. $250)			
BASIC (U.S. $250)			
COBOL (U.S. $250)			
FORTRAN (U.S. $225)			
Pascal (U.S. $225)			
Editor (U.S. $125)			

The prices quoted above are typical for quantity 1. New documentation is included with the software.

7. Would you be interested in multiple-copy licence and maintenance contracts for any of the above? (See details below) ______________________________

8. We are considering producing a User's Guide specific to the SuperPET. What sort of information do you think would be most useful? ______________________________

____

9. What other products or enhancements would you like to see from WATCOM? (Please describe) ______________________________

____

____

____

10. What are your comments on the WATCOM products to date? (a) Software, (b) Documentation, (c) Support

____

____

____

11. Please send me more information on: ______________________________

____

____

Name:	______________	Please mail this form to:
Title:	______________	WATCOM Products Inc.,
Company:	______________	415 Phillip Street,
Address:	______________	Waterloo, Ontario,
	______________	CANADA N2L 3X2
	______________	
	______________	Attn: Jack Schueler
	______________	

nnd forming of characters strings to numerics, Quad FI; error trap and handling with alternates, events, event messages, and event simulation (from APL2); workspace transfer fuctions; automatic workspace loading for applications in which the user has no knowledge of APL; multiple statements on one line; user-defined collating sequences to sort char arrays by different char sets; user can lock defined functions; major increase in speed of execution of primitive and user-defined functions; boolean, integer, and floating point numeric types with more efficient use of workspace; Quad MEMALLOC lets user allocate memory for PEEK, POKE, SYS. Quad CURSOR provides absolute control of cursor and senses location.

mBASIC: Formatted I/O with INPUT USING and PRINT USING with debit/credit/floating dollar, asterisk fill; free format structured control (for...next loop on one line--Yetch!); block labels on structure statements to quit from more than one level without flag (YAY!); short and long precision floating point; settable print zones; renumbering and move of subset of a program; optional traceoff and traceon in debugging; logging of terminal I/O to a file; LIST displays workspace time, date, name; LOCAL variables and arrays within functions and procedures; new matrix functions for sorting; mat inverse added; new trig functions COSH and LOG10; lower/uppercase translation; string replacement and translation.

COBOL: Supports most of ANS COBOL ('74), level 1 of modules Nucleus, Seq I/O, Relative I/O, Table Handling; parts of level 2 in above modules; plus full support of PERFORM, STRING, and UNSTRING verbs; improvements in speed and diagnostics; bugs fixed. No extensive new features.

FORTRAN: More closely adheres to FORTRAN 77. COMMON and DIMENSION statements added; char. support to FOR 77 standard; REAL and ATAN2 added; variable definition check optional; subprogram recursion no longer allowed; "A" format code revised to FOR 77 standard; char variables of fixed size, declared with length attribute. Existing programs affected by last item.

PASCAL: Faster. Proc and function names can pass as parms to other proc and fn. File random access improved, UPDATE opens for both I and O; SETNEXT sets next file element used; APPEND proc appends to previous file; "else" clause added for default in CASE operations; pseudo-random number generation.

MicroEDITOR: Line split; line join; new metachar "%#" substitutes for line number, allowing line numbering in workspace; "exit" substituted for "bye" (YAY!); get and put no longer change current filename (YAY!); current filename on the screen. We assume without confirmation that revised mED runs in all languages.

nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn

(*C*) 1984 ◇◇◇ *THE APL EXCHANGE* ◇◇◇ *STEVE ZELLER*

uuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuu

Late last year, I described how difficult it was to interact with a host computer from APL via the serial port (Vol. I No. 11, p. 163). I could see various applications that would enhance the usefulness of APL, but was frustrated in carrying them out. I concluded that only a ML routine would be fast enough to do the job. Loch Rose agrees. He has developed and is using (from mBASIC) some nifty code to do it! He kindly sent along his routines, documentation, and some examples in APL. I altered the APL code a bit and used Loch's ML routines, with the results described below (see the separate article, this issue).

Loch's strategy is built around two ML routines and a 2K buffer that live at the top of the APL workspace. The first sends a character string (80 characters or less, followed by a <CR>) to the serial port. The routine then "listens" until it either detects an incoming character or the waiting time exceeds a limit you have specified. The APL cover function for this is PUT_SERIAL, shown below. If the waiting time (time-out) parameter has been exceeded, you can assume that you have a problem at the serial port (properly initialized, modem turned on, etc.). A fairly long waiting time, such as 20 seconds, works well for some commands; a host computer typically takes its time to consider your request before sending characters back. When the host does respond, characters arriving at the serial port are placed in the buffer.

Loch's second ML routine extracts characterts from the buffer under three conditions: (1) the buffer is full, (2) you have received "Return" characters (up to five can be specified at one time), and (3) the wait time exceeds a maximum you specify. (You may PEEK to determine the cause.) The APL cover function for this ML routine is found below as GET_SERIAL. We initialize the serial port (at 1200 baud in this example) with INIT; SET_PROMPT may be used to set the return characters used by GET_SERIAL. Loch also provides a terminal facility which is not described here but is available in the sample WS.

```
      SERIAL
1200 96 0
      ∇INIT[⎕]∇
[  0]    R ← INIT PORTPARMS ;X
[  1]   →(3≠N←⍴PORTPARMS)/ERR
[  2]     0⍴PORTPARMS ⎕SYS 29952
[  3]     SET_PROMPT ⎕AV[⎕IO+10]
[  4]     → ¯1+R←1
[  5]   ERR:'WRONG NO. OF PARAMS'
[  6]     R←0
      ∇SET_PROMPT[⎕]∇
[  0]    SET_PROMPT CHARS ;⎕IO;N;PROMPTS
[  1]   ⎕IO←0
[  2]    N←⍴,CHARS
[  3]   PROMPTS←5⍴⎕AV[0]
[  4]   (PROMPTS[⍳N]←,CHARS) ⎕POKE 1 2⍴30197 30201
```

```
      ∆WAIT1,∆WAIT2
20 1
      ∇PUT_SERIAL[⎕]∇
[  0]    R ← PUT_SERIAL MSG ;⎕IO;N
[  1]   ⎕IO←0
[  2]   MSG←((N←80⌊⍴MSG)↑MSG),⎕AV[13 0]
[  3]   MSG ⎕POKE 1 2⍴30447+0,N+1
[  4]   R←∆WAIT1 ⎕SYS 30363
      ∇GET_SERIAL[⎕]∇
[  0]    R ← GET_SERIAL ;N
[  1]   R←⍳0
[  2]   →(0=N←∆WAIT2 ⎕SYS 30087)/0
[  3]   R←⎕PEEK 1 2⍴30529+0,N-1
```

I decided to use these routines to accomplish two tasks: 1) to log onto a host computer automatically, and 2) to interactively bring data directly into the APL WS from the host. In the first task, I specify: a) the message to be sent out the serial port, b) the message (or part of it) expected back from the host, and c) the number of time to look for the correct return message before you decide that something has gone wrong. An APL routine to prompt for this information is shown below. It stores the logon procedure in three global variables. To save space, I use character strings rather than matrices.

```
      ∇BUILD_PROC[⎕]∇
[  0]    BUILD_PROC ;ANS
[  1]   ⍝ BUILDS A ¨PROC¨ FOR LOGGING ON A MAINFRAME
[  2]    ∆MESSAGE←∆REPLY←''
[  3]   ∆TRIES←⍳0
[  4]   S1:'ENTER: MESSAGE TO SEND TO HOST'
[  5]     ∆MESSAGE←∆MESSAGE,⍞,∆TCNL
[  6]   S2:'ENTER: DESIRED REPLY TO MESSAGE'
[  7]     ∆REPLY←∆REPLY,⍞,∆TCNL
[  8]    S3:'ENTER: NO. OF TRIES TO FIND REPLY'
[  9]      ΔTRIES←ΔTRIES,⎕
[ 10]    S4:'MORE (Y/N)?'
[ 11]      →(~∧/(ANS←⍞)∊'YN')/S4
[ 12]      →('Y'=1↑ANS)/S1
[ 13]    'DONE'
```

To represent the sequence of commands, consider the following hypothetical table as an example:

Message to Send	Response	No. of Tries
AT	OK	2
AT EO	OK	2
AT DT9991234	CONNECT	10
	LOGON	10
logon me	PASSWORD	5
cryptic	DONE	5

The first three commands are modem-specific (HAYES). After checking to see if the modem is turned on, the commands turn off the local echo and initiate auto-dialing. If the modem responds with "CONNECT" then the host computer is on-line. This example assumes an ASCII system which requires a <CR> to get things started. These initial steps can be slow on some systems, so the APL program looks up to ten times for the proper response. Next in the table is a logon command, followed by an entry password. The character used to determine the end of a message in the buffer is a <CR>. Since the host may send a empty line or two before any message, or a line containing a message we do not expect, the number of tries is set to five. I don't completely search each response string from the host; in APL it simply takes too long. Instead, I just check to see if each character of the correct reply is in the response (the order could be entirely different). (For this reason, take some care in choosing replies.)

Once you have initialized the serial port and your procedure has been built, you may employ DIALUP to log on the host automatically. The relevant messages and replies are pulled out of their respective strings with the help of NEXTLOC, and CHECK_SERIAL loops for the correct response.

```
       ∇DIALUP[⎕]∇
[  0]     DIALUP ;I;I1;I2;N1;N2
[  1]    ⍝ROUTINE TO DIAL UP HOST COMPUTER AND ¨LOGON¨
[  2]    I←I1←I2←0
[  3]    S1:→(0=N1←I1 NEXTLOC ΔMESSAGE)/0
[  4]      →(0=PUT_SERIAL ΔMESSAGE[I1+⍳¯1+N1])/ERR1
[  5]      I1←I1+N1
[  6]    S2:→(0=N2←I2 NEXTLOC ΔREPLY)/0
[  7]      →(0=ΔTRIES[I←I+1] CHECK_SERIAL ΔREPLY[I2+⍳¯1+N2])/ERR2
[  8]      0 0⍴GET_SERIAL
[  9]      I2←I2+N2
[ 10]    →S1
[ 11]    ERR1:'MODEM/HOST OFFLINE'
[ 12]      →R←0
[ 13]    ERR2:'ERROR ENCOUNTERED DURING LOGON'
[ 14]      R←0
```

```
      ∇CHECK_SERIAL[⎕]∇
[  0]    R ← NTRY CHECK_SERIAL MSG
[  1]   ⍝ CHECKS SERIAL PORT FOR MESSAGE
[  2]   I←1
[  3]   S1:⍞←RESP←GET_SERIAL
[  4]     →(R←∧/MSG∊RESP)/OK
[  5]     →(NTRY>I←I+1)/S1
[  6]   OK:
      ∇NEXTLOC[⎕]∇
[  0]    R ← START NEXTLOC MESSAGE
[  1]   ⍝GETS LENGTH TO NEXT LOCATION OF <CR> IN STRING
[  2]   R←1↑((START↓MESSAGE)∊ΔTCNL)/⍳⍴START↓MESSAGE
```

Note that a great deal of looping goes on: programs written in other languages probably would result in better performance. APL, however, is well-suited for the next task--in which we bring data from the host into the WS in a simple, flexible way. Now that we're logged on, assume that a host program gives us access to the data. Next, assume that this program prompts for more input with an ASCII "?". Finally, assume that the host software allows us to print a series [in this example, GNP, when we give the command: P GNP (in uppercase ASCII)]. The host then responds with the series name and with the data. Our strategy is to send a command to the host and then to collect all the characters returned until the next prompt. This function is performed in APL by TALK, which employs PUT_SERIAL, GET_SERIAL and REASON. The latter checks to see if the last return from GET_SERIAL is caused by the receipt of a prompt. If not, we collect more data from the serial port. We remove nonnumeric characters from the response with CLEAN; and the result is returned as a numeric vector.

```
      ∇TALK[⎕]∇
[  0]    RESP ← TALK MSG
[  1]   RESP←⍳0
[  2]   →(0=PUT_SERIAL MSG)/NOANS
[  3]   S1:RESP←RESP,GET_SERIAL
[  4]     →(0=REASON-⎕IO)/S1
[  5]   →0
[  6]   NOANS:'NO RESPONSE'          |-----------------------------------
      ∇REASON[⎕]∇                     | REASON RETURNS ONE OF THREE VALUES
[  0]    R ← REASON ;⎕IO              | 0   -  BUFFER IS FULL
[  1]   R←⎕AV⍳⎕PEEK 30205+⎕IO←0       | 255 -  TIME OUT
      ∇CLEAN[⎕]∇                      | 1   -  EOT CHARACTER RECEIVED
[  0]    R ← CLEAN DATA               |-----------------------------------
[  1]   ⍝REMOVES UNWANTED CHARS FROM STRING BEFORE CONVERSION
[  2]   DATA←(DATA∊' .0123456789')/DATA
[  3]   R←⍎DATA
```

Are we ready? The sample session below shows how to converse with the host from the APL WS. After we first get the prompt, the ASCII command to print a series is sent to the host and 61 characters return. CLEAN converts them to floating point data and stores them in the APL variable, GNP. Matters are not always so simple, however. For example, a valid variable name may contain numbers (e.g., GNP72), or two data points may be separated by a <CR>. (Loch's routine strips out linefeeds.) But it's a start!

```
      ⎕AV⍳QMARK
64
      SET_PROMPT QMARK
      ⍴DATA←TALK '* ∇T*'
* ∇T*
61
      DATA

∇T*  3,171.500  3,272.000  3,362.200  3,436.200  3,550.100

      ⍴GNP←CLEAN DATA
5
      GNP
3171.5 3272 3362.2 3436.2 3550.1
```
∩∩∩

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

∪∪∪

FASTER BY AN ORDER OF MAGNITUDE... In issue 13, p. 234, we printed a note about handling PRG files with LINPUT, and said it beat a GET hollow. Associate Editor Terry Peterson went to work and rewrote the Waterloo patch for mBASIC we published last issue, which ran in 31 minutes. Terry's rewrite runs in 2.5 minutes, using LINPUT. Meanwhile, ye ed did it a different way with LINPUT, and got a patch which runs in 133+ seconds (2.2 minutes). Please observe that a straight copy of the mBASIC interpreter from disk to disk, using the mED's COPY command, requires 3 minutes 23 seconds--without patching. When we said LINPUT handles PRG files with dispatch, we meant it. Those who want a copy of both revised patches, please send a self-addressed stamped envelope to the editor. Curious, Waterloo?

REACHING WATCOM : Goodbye infoWAT & WATNEWS And hello WATCOM News, which replaces both former publications above. Those who subscribed will receive WATCOM News instead; subscription is $10 U.S. in the U.S. or Canadian in Canada, for one year; send checks or money order, not cash, to WATCOM News, 415 Phillip St. Waterloo, Ontario, Canada N2L 3X2. Address any communication regarding Waterloo software to WATCOM Products, Inc. at the address above. Note change from the old address on University Avenue. Do not write the University of Waterloo!

Prices, back copies, Vol. 1 (Postpaid), $ U.S. : Vol. 1, 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. 4: $1.25	No. 7: $2.50	No. 10:$2.50	No. 13: $3.75	

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

==

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

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

Name:______________________________ Disk Drive: _________ Printer:____________

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

[] Check if you're renewing; clip and mail this form with address label, please.
For Canada and the U.S.: Enclose Annual Dues of $15:00 (U.S.) by check payable to ISPUG in U.S. Dollars. DUES ELSEWHERE: $25 U.S. Mail to: Paul V. Skipski, Secretary, ISPUG, PO Box 411, Hatteras, N.C. 27943, USA.

SCHOOLS: Send check with Purchase Order. We do not voucher or send bills.

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

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

FIRST CLASS MAIL

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

First-Class Mail
in U.S. and Canada