The logo left was drawn by Brother James Kane of the Holy Cross High School in Waterbury, Connecticut, a member whose first letter arrived in the calligraphy of a medieval monk; we asked for a logo from his hand, and here it is.

NEW ASSOCIATE EDITORS

The Gazette is proud to announce three new Associate Editors: Terry Peterson, Associate Editor at Large: [mBASIC, mPASCAL, mAPL, 6502 Machine Language, and mFORTRAN]; Roy Busdiecker, Associate Editor at Large [6502 Machine Language, BASIC, and whatever else he's curious enough to peer into]; and Stephen Zeller, who will concentrate on APL. The first of Steve's articles on APL appears in this issue. Our previous Associate Editors [Gary Ratliff, 6809/6502 Asssembler and Robert Davis, mPASCAL] of course remain with us, though Bob's been seriously ill. We still need someone to handle mCOBOL and another hand at mFORTRAN. Many hands make light work, and we'll have a far broader point of view. Volunteers in mCOBOL and mFORTRAN please step forward.

ON BEING A BIT MULTILINGUAL

The editor has been dragged, screaming and kicking, into all the SPET languages, having a firm rule that the Gazette publishes only what's been tried, understood, and run without trouble. And so, a few words of advice to the readers: Please do not remain totally monolingual because you have a favorite language and are not interested in anything else. What is said in one language often applies to other languages. Examples: (1) the mBASIC screen dump printed in Vol. 1, No. 2, was adapted to APL by one multilingual reader. We print it this issue; (2) the set-time, set-date process discussed by P.J. Rovero in this edition applies, apparently, in all languages but APL; (3) the DOS utility printed in Vol. 1, No. 3 handles files in all SPET languages, including BASIC <4.0 and APL--if you capitalize BASIC filenames and enter APL filenames in lower case. You can read all directories from 6809 mode (you cannot from 6502). A DOS program in any 6809 language but APL will also work in all languages. You cannot handle BASIC 4.0 from APL easily since APL sends lower case Roman to the DOS, despite the upper case characters on the screen; though if you're good at Sanskrit, you can use the upper case APL symbols to substitute (not easy!); (4) the article on tabset by Dr. John Spencer, this issue, though written in mBASIC, provides POKE and PEEK methods to reset tabs from program in any language which will accomodate POKES and PEEKS; (5) what we say this issue about text-processing using the microEDITOR applies to every language, including APL--and to both WordPro and Wordcraft files. Confine yourself to one language and you'll miss a lot!

mBASIC turns out to be a lingua franca for the Gazette; most programmers understand it, and, unlike BASIC 4.0, it is highly structured and highly readable. If general material were presented in any other language (PASCAL? APL?), how many could read it?

WATCOM ANNOUNCES MICROPIP AND THE 6502 ASSEMBLY-LANGUAGE SYSTEM
(December infoWAT)

Waterloo Computing Systems, Ltd. announced availability of both packages, left, after our January issue went to press. From what we have learned, the 6502 package does for the 6502 what the present DEVELOPMENT package in SPET does for the 6809. The red herring says it allows you to develop software for other Commodore machines--8032, 4032, VIC 20, or the 64. It uses the mED, in 6809 mode. We've ordered it and will get an evaluation as soon as we can. MicroPIP (peripheral interchange program) provides utilities for common operations on HOST, DISK, and SERIAL devices. When we learned it also incorporated direct commands such as BACKUP and

FORMAT, we hurled a check pronto for a copy. Will report. WATSOFT Products, Inc., 158 University Avenue West, Waterloo, Ontario, Canada N2L 3E9, distributes. Prices: $75.00 for microPIP; $250.00 for the 6502 package.

If you don't subscribe to infoWAT, consider it. December has two helpful articles, one on Macros, and a second on data transfer between computers using the SPET serial port. Price: $10.00 for ten issues (U.S. in the U.S., Canadian in Canada), from infoWAT, PO Box 943, Waterloo, Ontario, Canada N2J 4C3. Four printed pages, and never an issue so far without useful information.

While on useful publications: MICRO magazine (MICRO INK, 34 Chelmsford Street, PO Box 6502, Chelmsford, MA 01824, $24.00 per year) has begun support for SuperPET. See an article on mBASIC by Loren Wright in Oct. '82, a comprehensive review of WordPro vs. Wordcraft in Nov. '82; one on the SPET character set by Terry Peterson in Dec. '82, followed by an article on mAPL, again by Terry Peterson, in Feb. '83. In addition, Dr. William Dial, who handles the 6809 bibliography for MICRO, asked for and now receives the Gazette so that our material may be incorporated. We're highly pleased to see some support at last.

FISHER SCIENTIFIC: MAINTENANCE & SALES

We've done business with Fisher for many years; the firm used to handle high-quality chemicals and lab equipment, and we always got good quality and good service. Now, we find that Fisher has expanded into much more complex gear, is a dealer for Commodore, and, more important, offers maintenance and repair services for Commodore computers. It's no surprise that a firm dealing in laboratory equipment must be into computers; or that the Commodore line was picked, considering that marvellous IEEE port and its adaptability to lab uses. This arrangement should plug one of the weak points in the Commodore line--service. We've talked to the maintenance people at Raleigh, N.C., and find them knowledgeable. We'd like reports on the service you get from Fisher. Here are U.S. and Canadian locations:

Atlanta	404 449 5050	Philadelphia	215 265 0300
Boston	617 391 6110	Pittsburgh	412 784 2600
Chicago	312 773 3075	Raleigh	919 876 2351
Cincinatti	513 793 5100	Rochester	716 464 8900
Houston	713 495 6060	San Francisco	408 727 0660
Los Angeles	714 832 9800	St. Louis	314 991 2400
New York City	201 379 1400	Washington, D.C.	301 587 7000
Orlando	305 857 3600		
Edmonton	403 483 2123	Quebec	418 656 9962
Halifax	902 469 9891	Toronto	416 445 2121
Montreal	514 342 5001	Vancouver	604 872 7641
Ottawa	613 225 6752	Winnipeg	204 633 8880

NEW PATCH FOR MICROBASIC

Waterloo Computing Systems Ltd. has forwarded a new patch for mBASIC, Verson 1.1 which arrived a little too late for our January issue. We've patched using it, and the 80-character string bug as well as the printer bug reported last issue have been fixed. The version to the left uses integers to save time, and runs in 36 minutes.

```
10 ! patch for MicroBASIC
20 open #2, "disk/1.BASIC,PRG", input
30 open #3, "disk/0.BASIC,PRG", output
40 x = peek(86)*256 + peek(87) + 4
50 y = peek(x)*256 + peek(x + 1) + 1
60 poke y, 0, 0
70 curr_posn% = 1 : p% = 1
80 call patch (39, 24*512+31)
```

The copy printed came straight off disk after we'd patched and tested

(cont. next page)

(cont. from previous page)

```
 90 call patch (0, 26*512+183)
100 call finish_up
110 close #2 : close #3
120 !
130 proc patch (new_byte%, address%)
140 for j% = curr_posn% to (address%-p%)
150   get#2, a$
160   if a$ = "" then a$ = chr$(0)
170   print #3, a$;
180 next j%
190 get #2, a$
200 print #3, chr$(new_byte%);
210 curr_posn% = address% + p%
220 endproc
230 !
240 proc finish_up
250 on eof ignore
260 loop
270   get #2, a$
280   if io_status = 2 then quit
290   if a$ = "" then a$ = chr$(0)
300   print #3, a$;
310 endloop
320 endproc
```

the new version. Again Waterloo has rendered aid and assistance quickly and we deeply appreciate the help.

As with the original patch, put a backup language disk in drive 1 and a disk to copy to in drive 0; then run the program. After trial of the patched version, scratch BASIC on the backup and copy the patched version of BASIC to the backup disk. The backup's now a new master, with correct programs on it.

* * *

APL DOS COMMANDS

Thanks to Roy Busdiecker, we print the last of the DOS commands this issue. We knew for some time that the APL tutorial disk had a function called 'APL.DOS', but hadn't figured out how to use it for lack of examples. Roy pointed out that the function contained a function (layers within the onion) entitled 'DESCRIBE', which illustrates the DOS commands. You get it by)LOAD APL.DOS, and then pressing PF3, which names the functions within APL.DOS. 'DESCRIBE' may be listed with ∇ DESCRIBE[⎕] <RETURN>, and is easily dumped to printer with the APL screen dump printed elsewhere in this issue. Left, below is an example of an APL 'immediate mode' DOS command; it works so long as 'APL.DOS' is in workspace, just as an immediate mode mBASIC procedure works if in RAM and called. Enter the command exactly as shown left, above. Do NOT use the the double quotation marks found in 'DESCRIBE'. (All DOS commands are summarized in Vol. 1, pp. 15-16.) The command above gives the new name of POKEPIX to old file POKEPIC on drive 0. Roy Busdiecker gives us considerable insight into the relationship of the APL and Waterloo fonts, and into the function 'APL.DOS' with the the short APL function we print at left, below--a direct APL command.

```
DOS 'R0:POKEPIX=POKEPIC' <RETURN>
```

Clear APL workspace with)CLEAR <RETURN> and enter the function (See Steve Zeller this issue on how to do it.) Then type: DOS <RETURN>, and it will change filenames exactly as the previous DOS command did. While the function is still on screen, enter two new APL functions, POKEDN (to put the Waterloo font on screen); then enter POKEUP, which returns you to APL font. Do this before you 'run' either function. Then 'run' them. You will immediately see why the command directly above works--the 'x' sign (multiply, not letter 'x') is an '=' sign in Waterloo; the proper ordinate (ASCII on SPET) is thus sent to channel 15 of the DOS; note that rho in APL (SHIFT R) sends to DOS the ordinate

```
∇DOS
[1] ('IEEE8+15.p0(POKEPIX×POKEPIC') ⎕CREATE 1
[2] ⎕UNTIE 1
[3] ∇
```

of the capital 'R' required for this DOS command. If you examine 'APL.DOS', you see the conversion to ASCII.

```
∇POKEUP                          ∇POKEDN
[1] ⎕AV[12] ⎕POKE 59468          [1] ⎕AV[14] ⎕POKE 59468
[2] ∇                            [2] ∇
```

After the information above came in, we received a letter from Dr. John C. Wilson of the Computer Systems Group at Waterloo (support from Waterloo is superb), which confirms Roy Busdiecker's observations, and adds the following:

```
DOS 'C1:LONGNAME=0:LONGNAME'
      (the problem)
     *      *      *

DOS 'C1:X=0:LONGNAME'
DOS 'R1:LONGNAME=X'
     (the solution)
```

A "known problem is that APL truncates (to around 30 characters) the lefthand argument of [] CREATE [Ed: see below], which is used in line 5 of DOS. This makes some operations fail. A common example is in copying files with long names [left, above]. A bypass is to use the following commands [left], instead." The solution copies the file as 'X' and then renames it. [Line 5 of function DOS, to which Dr. Wilson refers, follows: [5] ('IEEE8+15.' , C) ☐ CREATE 1]. You may dump both 'DESCRIBE' and 'APL.DOS' from the screen, after listing, with the APL dump, this issue.

DOS COMMANDS THE SIMPLE WAY:
THE DISCOVERY BY JIM SWIFT

```
g ieee8-15.NO:newdisk, id
```

Jim Swift, RR#3, Nanaimo, B.C. Canada, V9R 5K3, dropped us a note and said he could enter any DOS command in mPASCAL with the 'get' command printed left. A 'get' ???. It works. Curious, we tried ALL the DOS commands in ALL repeat ALL the languages except APL (in both Versions 1.0 and 1.1), and are delighted to report that the method works in all those languages without exception; we include the mED in mBASIC and mED in DEVELOPMENT. Do NOT put filenames in quotes. You sometimes get a 'FILE NOT FOUND' error, but the DOS commands are obeyed nevertheless. At last we have a standard way to enter the DOS commands (Vol. 1, pp. 15-16) in all languages but APL, using the mED, without mBASIC open and close statements or the cumbersome mCOBOL and mPASCAL programs we previously published. We hereby award Jim Swift the Serendipity Cup for 1983. To the good people at Waterloo: We doubt you intended a 'get' to work as it does, but for the love of simplicity, please don't change the method when you next update software!

ON NETWORKS AND MUPET
WITH SUPERPET

Jim Swift also reports he has three SPETS and one 8032 networked to a 4040 drive and a Qume Sprint 5 printer, with MUPET 2. He uses the 8032 for data base work, employing JINSAM and WordPro4+; in addition, there's an MCM 900 with two double-sided 8-inch disks, and a Chatsworth card reader (initial trouble getting that hooked up). Primary system use is APL. Jim offers advice and help to any members interested, either at 604-753-8969, or through his I.P. Sharp mailbox, SWIFT. We suspect he'll answer letters, too, if you're without phone or modem.

RELATIVE CURSOR CONTROL
THE EASY WAY
by Dan Horn

Using those chr$(n) things to move the cursor around is a pain. You can't remember what they mean; you can't read a program and figure out what's going on; and chr$(whatever) runs S-L-O-W in microBASIC, as it does in BASIC. If you want speed you must convert chr$(whatever) to a string variable. I've worked out a simple way; it's easy to write; takes less room; is easy to read and runs faster. You can spot a cursor command at a glance.

You ought to adopt this method (or one like it) as a Gazette standard. I use CAPITALS to make the commands stand out, and never use CAPS elsewhere in a program (except in print statements). See a capital letter, you know it's a cursor command. When two CAPS are used, they are the first letters of the words in the command:

```
U$ = Cursor Up                  DL$ = Delete Left (SPET repeat key)
D$ = Cursor Down                DR$ = Delete Right (SPET delete key)
R$ = Cursor Right                T$ = Tab over one tab setting
L$ = Cursor Left                CS$ = Clear Screen
H$ = Home                       EL$ = Erase Line
G$ = One Space                  CR$ = Carriage Return
```

Why that G$? Well, I used SP$ for a space for a while, but I saw you were using g$ in the GAZETTE, and I wondered why. Then it hit me! A G-string on a great girl I know covers the minimum (I mean MINIMUM!), and one space is minimum. Is that the reason you use it? [Ed: No, but with a mnemonic like that, who can forget it? Spangles, Dan?]

All you need in any M-BASIC program is two lines of code, like below. I've got them on disk as 'start' and call them back as the first two lines of any new program. The system is as as easy to use as the old PET commands (easier if you want hard copy). See the wee example which follows the 'canned' code lines. It runs 15% faster than the same program using chr$(whatever).

```
10 H$=chr$(1):DR$=chr$(4):EL$=chr$(6):R$=chr$(7):L$=chr$(8):T$=chr$(9)
20 D$=chr$(10):U$=chr$(11):CS$=chr$(12):CR$=chr$(13):DL$=chr$(127):G$=' '
30 !
40 print CS$;'Clear Screen, go to margin and double space all text.';D$
50 print T$;T$; 'LET US CENTER ALL CAPTIONS AND TITLES.';D$
60 print T$; 'And Indent All Instructions One Tab';D$
70 print rpt$(G$,30); 'AND CENTER WARNINGS!';D$
80 print 'Or Delete the Warning';CR$;U$;U$;U$;EL$
```

The manuals say to use spaces within quotes to move text right, as on the left:

```
print '                    CENTER CAPTION.'
```

If you want to save on memory, use T$. The line on the left uses 42 bytes. T$ uses a third less. Even rpt$(G$,20) uses less memory, but runs slower. The fastest method is T$--and it is faster to use up to three separate commands like U$;U$;U$ than rpt$(U$,3). After that, rpt$ is shorter to write and as fast. The semicolons I use between commands run a little faster than '+'. I wish there was a way to set tabs from program, because T$ is mighty handy. [Ed: See Dr. Spencer's article, this issue. Dan makes a square wheel round. We've used his system since his article arrived. It is all he claims: short, easy to write, easy to read, and fast. Comments, please. We'd like to make it a Gazette standard if the readers agree.]

SETTING SCREEN TABS AND USING CHR$(9) IN MICROBASIC

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

Unlike PET BASIC, the Waterloo MicroBASIC 'print tab(n)' function in program erases all text it passes over; we modify a table entry only by rewriting the entire line. We may evade the problem with (1) a cursor-right (rpt$(chr$(7),n)), which does not erase, lets us translate easily from BASIC to MicroBASIC, but runs slowly; or we can (2) print with the 'cursor' function, as previously noted

in the Gazette. There is a third, often simpler solution.

MicroBASIC implements the TAB key in immediate mode, and in a program 'tabs' with chr$(9). Both skip the cursor from its current position to the next pre-set tab without erasing text. If the first tab stop is set at column 8, a tab command from left margin moves the cursor to column 9; successive tab commands move cursor to successive tab positions, exactly as on a typewriter.

When SPET powers up, the 10 available tabstops are preset at intervals of 8, starting at the left margin of the screen (i.e., 0, 8, 16...72). We may change the stops in the microEDITOR (see manual); upon return to mBASIC, these new stops remain in effect--a virtue and a nuisance, for while the new stops may work for a specific table or task, we may well need other tab settings for another task in the same program. (As indeed we do in the demonstration programs following.)

Though mBASIC provides no direct 'tabset' facility, we may PEEK and POKE the settings easily once we know where to look in memory. SPET stores tab stops in successive two-byte memory locations starting at 270 and ending at 288-289. Ordinarily, only the odd-valued addresses (271, 273, etc.) contain tab stop values.

```
120 print chr$(12); 'ADDR','TABSTOP'
130 for i% = 271 to 289 step 2
140   print i%, peek(i%)
150 next i%
```

To see current values, run the program to the left. We may change values by a POKE to odd-valued memory locations. For stops in increments of 10, POKE the values as in the example below. As with any POKE, we must be most careful when we enter the statement. Zero must appear between each value poked and the next, since the extended POKE in Waterloo mBASIC fills each successive memory location starting at, and after, the pointer or argument, which is 271 in the example.

```
poke 271,0,0,10,0,20,0,30,0,40,0,50,0,60,0,70,0,0,0,0
```

The statement POKEs a zero tabstop first into the odd location 271; then places a zero in each even-valued address (high-order byte of the tab stop), and pokes the value of the stop in the low-order byte. SPET positions the tab stops with the full two-byte value, so it is possible to set 'giant' tabs (see below). After the POKE above, we find a 0 in locations 287 and 289. This value, being lower than previous ones, is ignored by SPET when a TAB command is issued. SPET always executes tabstops in ascending order.

With the tabstops above, any TAB command past the last tabset of 70 (cursor at 71) wraps the cursor to the start of the same line (provided a tabstop has been set at the left margin, or 0. Lacking a zero setting at left margin, the cursor wraps to the first set tabstop). If we now 'poke 287,80', and TAB to it, the cursor wraps to the start of the next, lower line (the cursor always goes one position past the set tabstop). If we TAB again, the cursor runs to the next higher stop, at 10 (cursor on column 11).

Next, 'poke 287,95', and TAB across the screen. The cursor comes to rest at position 16 on the next line, since the tabset of 95 exceeds 80 by 15 (+1). We cannot POKE another, higher value into memory location 289, for SPET recognizes only one tabset over 80. Ten tabstops are available; do not attempt to POKE values beyond address 289.

In a program, we POKE the desired tabset positions and TAB to them in print

statements with chr$(9). We may build tabs into strings with chr$(9), as is done in the demonstration programs which follow. (Note: a reset of tabstops does not affect the size of the mBASIC 16-space print zones invoked with commas between items in print statements.)

In 'tab demo#1', following, the last tabset of 90 in line 120 forces a carriage return and aligns data columns. Note that the program resets normal tabstops before ending, as does 'tab demo#2', which creates a giant A$ containing embedded TAB characters; it outputs the whole string with nothing more than a 'print A$' at line 300. Two different screen formats are included to show how easy it is to change the screen display. Simply move the '!' from line 240 to line 230 to see that change. 'Tab demo#2' takes a while to run; be patient. Since it resets tabstops to normal positions in line 310, we see the effect of no automatic carriage return on A$ if we print it in immediate mode as soon as the program has run.

```
100 ! tab setting demonstration #1 for SPET : title : 'tab demo#1'
110 ! PEEKs out all tab set locations 270-289 in paired columns
120 poke 271,0,0,9,0,25,0,49,0,65,0,90 ! set tab stops for title
130 print chr$(12);rpt$(chr$(9)+ "ADDRESS" + chr$(9) + "CONTENTS",2)
135 poke 271,0,0,10,0,28,0,50,0,68,0,90
140 for i% = 270 to 289
150   print chr$(9); i%;chr$(9); peek(i%);
160 next i%
170 print : print
180 poke 271,0,0,8,0,16,0,24,0,32,0,40,0,48,0,56,0,64,0,72
190 stop   ! line 180 restores the normal tabs stops on exit

200 ! tab setting demonstration #2 for SPET : title : 'tab demo#2'
210 ! Shows the ordinal value, corresponding character and reverse
220 ! field character.
230 poke 271,0,0,6,0,22,0,38,0,54,0,70,0,86 ! 5 columns
240 ! poke 271,5,0,13,0,21,0,29,0,37,0,45,0,53,0,61,0,69,0,85 ! 9 columns
250 print chr$(12);A$=""
260 for i% = 14 to 127
270   i$ = value$(i%)
280   A$ = A$+chr$(9)+i$+rpt$(" ",4-len(i$))+chr$(i%)+chr$(i%+128)
290 next i%
300 print A$
310 poke 271,0,0,8,0,16,0,24,0,32,0,40,0,48,0,56,0,64,0,72
320 stop
```

If we format a table with TAB commands, we may selectively replace any element in the table: put the cursor on the correct line and TAB to the location of the element. Despite any change in tabsets after a program is written, we will always reach the correct position to rewrite the element.

```
poke 272,7,206
print chr$(1);chr$(9);'A';chr$(1)
```

We may set giant tabs (up to 1998) by setting both high and low bytes, as in the example to the left, above. It prints the letter 'A' at screen position 1999; the total tab distance is 256*peek(272) + peek(273) [256*7 + 206 = 1998].

Although we focus here on on mBASIC, we should be able to tab in the same manner in any SPET language which implements peeks, pokes, and the tab command. -End-

Ed: With what has been published previously, the two articles above pretty well wrap up cursor and printing control in mBASIC. Embarrassed by riches, we have

absolute cursor control, relative printing control per Dan Horn, the old tab(n) command, direct tabbing ala Spencer, and the comma print zones. The question in mBASIC is not how to control printing, but which of the numerous methods is best fitted for a particular job. Frankly, the other languages seem somewhat primitive in this respect. Horn's relative control method is a jewel, as is Dr. Spencer's method of setting tabs in program. The long POKEs of the latter are dangerous, however, if you make an error, and slow to write. Having to program a large number of tables, we wrote a 'tab set' immediate mode procedure to generate the POKEs if you specify the stops. To use it, format a sample of the table you want on line 7 or 8 of the screen; then 'call tabset'. It prints an index to screen positions on line 6, & a reverse field 'T' at each tabstop as you set it, plus the value of the tabstop. You then see the tabstops you set and the material to be tabstopped, at a single glance.

```
50000 ! title: 'tabset'
50010 proc tabset
50020 EL$=chr$(6)
50030 D$=chr$(10):L$=chr$(8):R$=chr$(7)
50040 H$=chr$(1):T$=chr$(9):print H$;
50050 for ii = 1 to 4  ! Clear top of screen
50060   print EL$;D$;
50070 next ii
50080 print EL$;rpt$(T$,4);"Index to Tabstops"
50090 for ii = 0 to 75 step 5  ! Print index
50100   x = cursor(400+ii) : print ii;
50110 next ii
50120 print H$; "Tab Stops Entered at: "
50130 for ii = 1 to 10 ! Set and mark stops.
50140   x = cursor(241)
50150   print EL$;"Enter Tabstop No.";ii;
50160   input ' ',nn(ii)
50170   ! Vary trapline below at will
50180   if nn(ii)<0 or nn(ii)>95 then 50140
50190   xx = cursor(80+nn(ii))
50200   if nn(ii)=0 or nn(ii)> nn(ii-1)
        then print nn(ii);D$;L$;L$;
50210   if nn(ii) >=10 then print L$;
50220   print chr$(212) ! Reverse 'T'
50230 next ii
50240 for ii = 1 to 19 step 2 ! Poke stops
50250   kk = kk + 1
50260   poke 270 + ii, nn(kk)
50270 next ii
50280 xx=0:kk=0:print D$;" ALL TABS SET "
50290 endproc
50300 !
50310 ! Reset tabs: 'tabreset'
50320 proc tabreset
50330 for ii = 1 to 19 step 2
50340   poke 270 + ii, kk
50350   kk = kk + 8
50360 next ii
50370 kk=0 : print " ALL TABS RESET "
50380 endproc
```

Because resetting tabs is as much of a chore as setting them, we include a 'tabreset' procedure which resets tabs to Waterloo default values. If you want a procedure which sets tabs in any chosen and unvarying increment, you'll find that a simple rewrite of tabreset won't solve the problem. We've written a procedure which lets you set tabs at any increment from 1 on up. If space allows, we'll print it at the end of this issue; if not, next time.

A FAST REVERSE FIELD PROCEDURE FOR PHRASES OR COMPLETE STRINGS

We've received a number of procedures for printing strings in reverse field, most of which are pretty slow because they concatenate the string and then return to the main program to print it. You need not do either for long strings; it's far faster to print from the procedure without concatenation. On short words or phrases to be included in a print statement, it's easier to concatenate before you print. The speed emerges from the one-line statement at 220, and from use of integers.

```
100 ! 'reverse all' : a demo program in mBASIC
110 print chr$(12): z%=128 : p%=1
120 rvsphr$ =" EXECUTING " : call reverse
130 print " Program is Now ";phr$(1);" a Reverse Field Word"
140 rvs$= " Full String is Reversed, Printed in Procedure "
150 x = cursor(410) : call reverse
160 stop
170 !
180 proc reverse
190   y% = len(rvs$) : q% = len(rvsphr$)
200   if y%
210     for i% = p% to y%
220       print chr$(ord(rvs$(i%:i%))+z%);
230     next i%
240     print
250   else
270     m = m + 1
280     for i% = p% to q%
290       a$ = chr$(ord(rvsphr$(i%:i%))+z%)
300       phr$(m) = phr$(m) + a$
310     next i%
320   endif
330   y% = 0:q% = 0:rvsphr$="":rvs$=""
340 endproc
```

The procedure to the left will print words, phrases, or full strings in reverse field more rapidly than anything else we have seen. All full strings to be reversed are called 'rvs$'; phrases or words, 'rvsphr$'.

A procedure call prints rvs$ quickly; words or phrases are reversed by a call before the print statement in which they are used. Each word or phrase, upon reversal, becomes phr$(m), where 'm', as written, can have a value of 1 to 10. Should you want more reverse field words, dimension the array for more.

Note that the procedure automatically determines whether to provide a reverse phrase or a full reversed string from the initial values of y% or q%, and that the cursor is positioned in the main program with the handy x=cursor(n) command.

A LINE DUMP FROM SCREEN TO PRINTER FOR APL

George Cordahi, of the Civil Service Commission, Room 204, Frost Bldg. S., Queen's Park, Toronto, Ontario, Canada M7A 1Z5, forwarded pages of useful general APL functions which we sent to Steve Zeller, Associate Editor for APL, to evaluate and to include in his forthcoming series. One of the functions, a line dump from screen to printer, proved so useful that we publish it, left, in its ENTRY format; we used it to dump itself as printed. When you call it with with CDUMP <RETURN>, it flashes ENTER QUIT TO STOP ten times. The cursor then homes. Press <RETURN> for each line dumped. CDUMP stops when it reads the line QUIT.

```
∇CDUMP
[  1]  ⍞←⎕TC[4]  ⍝ DIABLO AND 8300P
[  2]  T←0
[  3]  F1:SZ←⍴(⍞←'ENTER ''QUIT'' TO STOP')
[  4]  ⍞←⎕TC[SZ⍴1]
[  5]  ⍞←⎕TC[8]
[  6]  →((T←T+1)<10)/ F1
[  7]  'IEEE4' ⎕CREATE 3
[  8]  ⍞←⎕TC[6]
[  9]  D1:LINE←(LINE∈⎕AV[14+⍳113])/LINE←,⍞
[ 10]  →(∧/(¯4↑LINE)='QUIT')/D2
[ 11]  (⎕XR LINE) ⎕PUT 3
[ 12]  →D1
[ 13]  D2:⎕UNTIE 3
[ 14]  ∇
```

This dump is for Version 1.1 and for printers such as DIABLO or Commodore 8300P, which will backspace and overstrike. It will not work on the EPSON MX80 F/T P2. A slight change allows it to be used on the EPSON, according to George.

```
[ 11]  ⎕AV[(⎕AV⍳LINE),14] ⎕PUT 3
```

For the EPSON, change line [11] as shown at left. This provides overstruck characters and proper EPSON linefeeds (if the modified version is used on DIABLO, it will not print overstruck characters, and it double spaces

the printout). Note: printers which need to backspace to overstrike, such as Spinwriter, Diablo, and 8300P, must receive APL characters in XR (External Representation) format.

You are free to edit or revise the screen before dumping, so long as you do not press <RETURN>, as with the mBASIC dumps. George admits to a direct translation of the mBASIC dumps previously published. We're sure somebody will now write one of those famous APL one-liners to do the same job. We'd be happy to have it. But until that one-line gem comes along, we have a way to save the screen in APL to hard copy. (If you have something to dump on screen, and CDUMP is not in workspace, you have two options: (1) if there's room,)COPY CDUMP into active workspace; or, (2) if there's no room,)SAVE workspace to disk,)CLEAR it, and)LOAD CDUMP. If you're careful, what you want to dump is still on screen.

Anent stopping dumps on 'QUIT', Dr. John Spencer asks why we don't stop on the words 'DUMP' or 'CALL DUMP', which would be simpler and shorter. Why don't we? A case of the stupids; the simple things are not easy. We've changed all our dumps to conform to the suggestion. Simple and easy.

ON BASE II MODEL MST PRINTERS — P.J. Rovero (address below) notes that not all such printers have the APL character set, but that owners with the buffer option can define and install an almost complete set. He has a program which runs in 6502 mode and creates any type of character set (Base II printer only); it's free to any member who sends him a disk and a self-addressed, postpaid mailer (4040 format only). Disk will have a sample APL character set on it; don't expect full overstruck characters. An earlier version was published in COMPUTE! (Feb '82).

MISSING CHARACTER GENERATOR — Donald Momberg's mystery of a missing Waterloo font is solved: both Walt Kutz of Commodore and Dr. John C. Wilson of Waterloo advised that someone probably had removed the 4K ROM generator and substituted a 2K generator containing only the old Commodore fonts. So it turned out. Watch your dealer and repairmen if your boards go back for repair. Don says the generator is in UA3; best mark it before it leaves home!

SET-TIME, SET-DATE FROM mFORTRAN

by P.J. Rovero
SMC Box 1610, Naval Postgraduate School, Monterey, CA 93940

[Ed. Note: The methods set forth in this article apply also to mBASIC and to mPASCAL; we publish programs in both languages in the pages following, this issue.]

The Waterloo software 'time' and 'date' functions are different than those in Commodore BASIC, which treats ti$ as a reserved variable which can appear on either side of an equality (ti may not so appear), as: ti$ = '12304500'. This allows you to assign a value to ti$ within a BASIC program.

The Waterloo interpreters treat time and date information as functions, which may appear only on the right side of an equality. You may thus assign the value of time or date to another variable, but you cannot assign a value to the time or to the date within mFORTRAN, mBASIC, mCOBOL, or mPASCAL. You can, of course, set time or date while in the microEDITOR in any of these languages. APL has a time-setting facility, while mPASCAL has no instrinsic functions for retrieving the time or date.

The solution to this problem is both elegant and educational. The setdate_ and settime_ routines are part of the SuperPET system library (see Chapter 8, As-

sembler Manual). The routines unfortunately do not accept parameters in a fully consistent fashion within the various languages. All the Waterloo interpreters (except mBASIC) contain an enhanced SYS function which allows parameter passing. The solution I provide is in mFORTRAN, and can be adapted to the other languages (though with some difficulty in mBASIC).

You find the addresses of the system library routines in the files 'watlib.exp' and 'fpplib.exp', on the language disk, using the 'get' command with the micro-EDITOR ('old' will not work). The Assembler Manual provides the number and type of parameters required, along with the return value (if any).

Listing 1, below, shows the subroutine settime. The system routine settime_ requires four consecutive bytes be received, with the integer values of hours, minutes, seconds, and 'jiffies' (1/60ths of a second). If you poke these values to the locations shown, you use a bit of memory just 'above' the screen. Addresses and values greater than 32767 are expressed as negative numbers, because negative integers in Waterloo Fortran are stored in two's complement form with the MSB set (and look like normal integers greater than 32767). Calculate the two's complement (the negative number) by subtracting 65536 (64K bytes) from the actual value. mFORTRAN quite insistently rejects integers greater than 32767. Note also the form of the SYS function. It can appear only on the right side of an equality in mFORTRAN, where it is a function, not a command, as in Commodore BASIC.

mFORTRAN Listing 1 (P.J. Rovero)

```
settime (hybrid fortran/machine language)

Invoke from mFORTRAN with:
    call settime (i1, i2, i3, i4)
    where i1=hours, i2=minutes,
          i3=seconds, i4=jiffy

settime_ is a system library routine at
    $b0f9. It must be set as the two's
    complement because it is larger than
    32767 if decimals are used.


stash is free space RAM just above the
    screen RAM at $87f0. If decimal is used
    it must be in two's complement.

subroutine settime (hour, minute, second, jiffy)
   integer hour, minute, second, jiffy, settime_
   settime_ = -20231
   stash = -30736
    x = poke1(stash, hour)
    x = poke1(stash+1,minute)
    x = poke1(stash+2,second)
    x = poke1(stash+3,jiffy)
    q = sys(settime_, stash)
end
```

* * *

[Ed: SPET, like most Commodore machines, won't handle integers larger than 32767. See separate article, this issue. The owners of the new 64 find that when they ask ?fre(x), values above 32767 return as a negative numbers; if subtracted from 65536, the result is free memory. In SPET, a poked negative address works as well as the positive value does, but we now know that direct hex pokes or direct hex addresses are far simpler to use--and as fast as doing the job with negative numbers. In the listing, left, settime_ can have the hex address $b0f9. When we asked Associate Editor Terry Peterson for his views, he allowed he had a personal abhorrence of negative addresses, and sent back a revised subroutine with hex addressing. Note how much shorter it is: (next page).

When we sent a note to P.J. about use of hex, he wrote, 'I didn't think of it at the time--I guess that is what the Gazette is all about.' Indeed it is. We learn a great deal from each other. The

```
program timtest  (Terry Peterson)

print, time()
call settime (10,33,00,00)
print, time()
end

subroutine settime(ihr,min,isec,jif)

  character newtime
  newtime = char(ihr)//char(min)//char(isec)//char(jif)
  q = sys (cnvh2i('b0f9'),newtime)
  return
end
```

editor was all heated up on the use of negative integers and the speed they gave when Terry reined us in with a few examples of good hex application. Two approaches to the same problem shed light.

The intrinsic function 'cnvh2i', which converts hex to integer characters is used by Terry to get to settime_ directly. Hex does have its virtues! The rest of P.J.'s article follows.]

Listing 2, below, shows the program setdate; the only parameter passed is the date string. The method is quite straightforward. No muss, no fuss!

Both of these subroutines can be included in any of your mFORTRAN programs that rquire current time and date information. They can be adapted to other SuperPET interpreters. And, most importantly, they can encourage you to explore beyond the boundaries of the interpreters and to use the powerful routines in the system libraries. -End-

mFORTRAN Listing 2

```
* program setdate

    print, date()
    call setdate('Mar 25 1983')
    print, date()
end

subroutine setdate(newdate)
   character newdate
   q = sys (cnvh2i('b0f3'), newdate)
end
```

* * *

NOTES:

Only 11 characters are allowed in the date line. If spaces lie between characters, quotes must enclose all.

See p. 53, Systems Overview manual.

Note that the SYS function is not implemented consistently in SPET. In mBASIC, for example, Terry Peterson notes that only one parameter may be passed in a SYS call--the address of the called routine. In mBASIC, he uses a brief routine in machine language to 'condition' the D register for the settime call. The procedure below, written by Terry, passes parameters and sets 'time$' in mBASIC. Before you make your call to the procedure, print 'time$' in immediate mode; then reset time by a call; then print 'time$' again to see the effect. You can set 'time$' from program with the procedure, and of course you can print 'time$' in program any time with a print statement. We received two mBASIC procedures on the same day, one written by P.J. Rovero, and the second by Terry Peterson, remarkably alike though each was written without the knowledge of the other.

```
10 proc settime (h%,m%,s%,j%)
20   ! First put machine code in high memory:
30   poke hex('87f0'),hex('cc'),hex('87'),hex('f6') ! LDD #$87f6, set pointer
40   poke hex('87f3'),hex('7e'),hex('b0'),hex('f9') ! JMP $b0f9 goto settime
50   ! Now poke new time string just behind.
60   poke hex('87f6'), h%,m%,s%,j%
70   ! Go do it.
80   sys hex('87f0')
90 endproc
```

The procedure to the left is called with the hours, minutes, seconds, & jiffys as parameters in the call:

'call settime(15,35,0,0)' sets time at 15 hours, 35 minutes, 0 seconds and 0 jiffys (which Waterloo calls 'ticks').

If you're as curious as we were about the anatomy of the machine code: line 30 puts into address $87f0 the 6809 instruction code ($cc) to load the D register with a pointer ($87f6) in the next two bytes. Three bytes having been used, line 40 continues at $87f3 with 6809 instruction code ($7e) for a 'jump' to $b0f9, the settime_ procedure. When settime_ is called, it apparently (we say apparently because we haven't traced the code) looks at the D register for the address of the setttime string, which has been poked (in line 60), and starts at $87f6. (Note: you'll find the instruction codes for the 6809 in Appendix D of Lance Leventhal's book on the 6809, recommended by Gary Ratliff last issue.) Having this material in hand, we sat down and wrote a setdate procedure which works right well. You'll find the library routine 'setdate' at $bf03 (see the library list, watlib.exp, on the language disk, Version 1.1).

```
 10 proc setdate(dyte$) ! title setdate-b
 20   poke hex('87f0'),hex('cc'),hex('75'),hex('31') ! LDD #$7531,set pointer
 30   poke hex('87f3'),hex('7e'),hex('b0'),hex('f3') ! JMP $b0f3 goto setdate
 40   for ii% = 1 to len(dyte$)+1 ! The 1 ends dyte$ with 0. Don't remove!
 50     poke hex('7530')+ii%, ord(dyte$(ii%:ii%))
 60   next ii%
 70   sys hex('87f0')
 80 endproc
```

The setdate_ function in SPET wants a string; you have to give it one (or the ASCII equivalent). Call setdate with the parameter (left). Function 'setdate_' will not use more than 11 characters, no matter how many you enter. It will use fewer. That's why the ending 0 (line 40) is required. Note we had to change the memory location from $87f6 to $7531 (high RAM), because the 11th character always was clobbered by a reverse field symbol. Before you call setdate, ask: ? date$; make the call; then ask: ? date$ again to see the change.

```
call setdate('Mar 25 1983')
```

Since Terry was barely warmed up by the time he finished settime, above, he rolled out a program with a settime/gettime pair for mPASCAL, below. We also received (again in the same day's mail) a settime from P.J. Rovero for mPASCAL. Since Terry's version includes both settime and gettime, we print his version.

```
program time_test  (input, output);      (* Listing for mPASCAL *)

type tymtype = packed array [1..4] of char;
var hours, mins, secs, jiffys: integer;

procedure set_time (hrs, min, sec, jif:integer);
(*set system clock to the time specified by arguments of call*)
  var new_time: tymtype;
  begin
     new_time[1] := chr(hrs);
     new_time[2] := chr(min);
     new_time[3] := chr(sec);
     new_time[4] := chr(jif);
     sysproc (11*4096+15*16+9, address(new_time))
  end;
```

[Continued, next page.]

```
procedure get_time (var  hrs,min,sec,jif: integer);
  (*returns system clock time as 4 integers*)
  var temp: tymtype;
  begin
    (*call gettime_ ($b0fc) *)
    sysproc (11*4096+15*16+12, address(temp));
    hrs := ord(temp[1]);
    min := ord(temp[2]);
    sec := ord(temp[3]);
    jif := ord(temp[4]);
  end;

  begin
    get_time (hours, mins, secs, jiffys);
    writeln (hours:2,':',mins:2,':',secs:2,'.',jiffys:2);
    set_time (10,33,0,0);   (* call to set time at 10:33:00.00 *)
    get_time (hours, mins, secs, jiffys);
    writeln (hours:2,':',mins:2,':',secs:2,'.',jiffys:2);
  end.
```

This program writes the old time and then the new, as set in the set_time call. Terry comments, 'The pair aren't beautiful, but they work.' They do indeed. Our thanks to P.J. and to Terry for some useful tools.

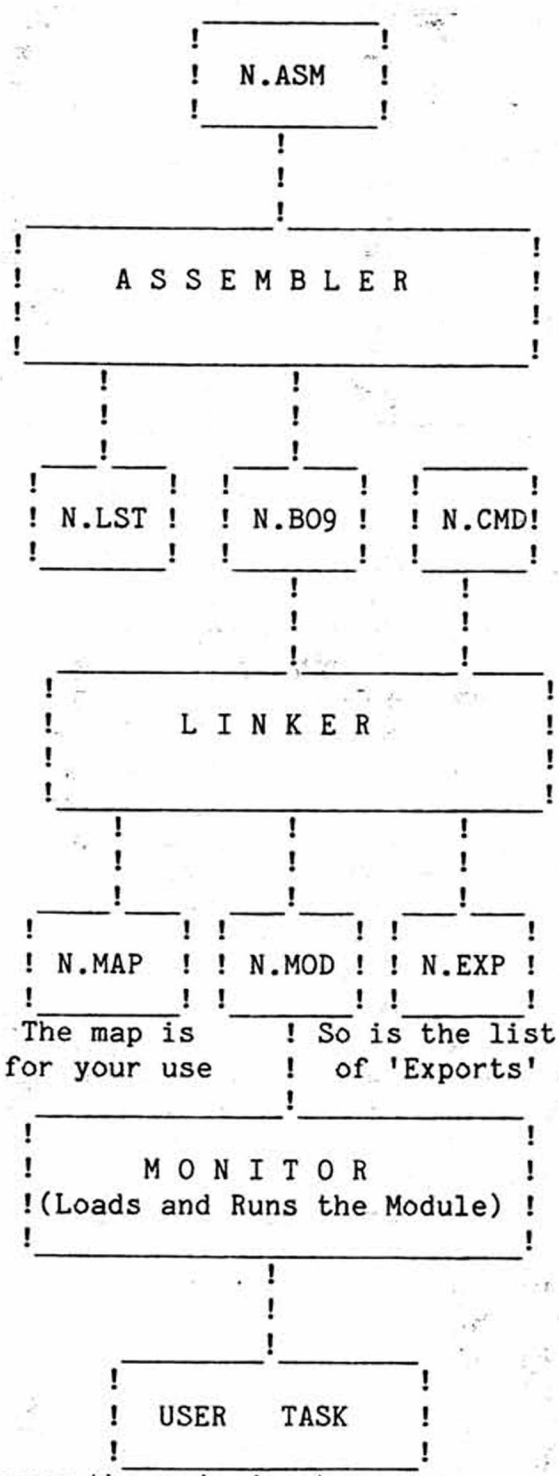
MARQUEE CONTEST WINNER Curses! Foiled again! When we announced a marquee contest in the December issue, it seemed obvious that you could write one by printing a character at right margin and deleting a space to its left to pull the string left--and that data statements (not arrays) would best store strings. So we gussied up a marquee with an array and no deletes, & with fiendish delight looped with a goto (forgot real evil: some gosubs). All that deception failed. Nobody used arrays; everybody deleted. Below, the winner, written by Terry Peterson, which runs TWO marquees so fast Terry had to slow it down with a delay loop--take it out at peril to thine eyeballs.

```
 10 ! marquee_n. Entry by Terry Peterson Dec 21
 20 print chr$(12):row1_end=8*80:row2_end=11*80:up_del$ = chr$(11)+chr$(4)
 30 c = cursor (661) : print "The marquee does not affect lines below"
 40 c = cursor (741) : print "(or above). When tired of this, press STOP."
 50 loop
 60   read d$
 70   if d$<> ""
 80     d$ = d$ + " . . . . " ! Delete this line to run strings together
 90     for a%=1 to len(d$)
100       if cursor(row1_end) then print d$(a%:a%);up_del$;
120       if cursor(row2_end) then print d$(a%:a%);up_del$;
130       for ti=1 to 10  ! Speed readers, take this
140       next ti         ! delay-loop out.
150     next a%
160   else
170     restore
180   endif
190 endloop   ! Following are strings to be displayed & terminating '""'
200 data "Next Week, We Open the SuperPET Follies, Starring Walt Kutz"
210 data "Walt Recites 'The Boy Stood on the Burning Deck', Sings, Dances "
220 data "The Editor Uses GOTOS"
230 data ""
```

BITS BYTES & BUGS

by: Gary L. Ratliff

The first order of business is to debug the example program from the last issue. The error was hinted at in my closing remark: "Bye for now." The correct command to leave the editor is: bye. The correct exits from the programs used in the development system are: bye to leave the EDITOR, q to leave the MONITOR, and RETURN to leave the ASSEMBLER or the LINKER.



As many of you have now realized, the DEVELOPMENT system for the 6809 processor is quite complicated. It includes the editor, assembler, linker, and the monitor programs. The diagram at the left depicts the relationship of the files and programs in the system. Let us now examine these files. The LST file is found on page 13 of the manual and won't be repeated.

```
EX1.MAP
Root  :
ex1.b09 = 1000 - 1005 .0006
Length of Module  = 0006
EX1.B09
Errors 0
$0005    ; Length
Object
8661b780003f
```

From the above listings the purposes of the many files created by the development system become known: The LST file incorporates the ASM file which we created with the EDITOR and a listing of the program along with the translation of the mnemonics into their equivalent hexadecimal instruction. LDA #'a is translated into 86 61. The 86 is the hex code for immediate load of the 'a' register. The 61 is the ASCII equivalent of the letter: 'a'. The MAP file tells where the assembled code is to be placed. The b09 file contains the length of the file as well as the object code. However, its most important information is the error count. If this count is not zero the linker will not produce the MOD file which is a file which may be executed either from the monitor or from the main menu. To prove this for yourself get the file ex1.b09 and use the EDITOR to change the error count from 0 to 1. You will find that the linker will refuse to process the code due to errors on assembly; the error being the altered count.

You may have thought that the task of placing a single character on the screen was trivial. However, the process of moving text from one area in memory to another is one of the most frequently performed tasks in any computer. Therefore, for our example program of this installment, let's expand; instead of

sending a single character, we'll send an entire message.

The assembly code to perform this task is presented below. As promised each example will contain at least one error for you to find. You were able to debug the code in the first installment so put on your thinking caps and find the error in this code.

The most common convention is to signal to the computer that it has reached the end of any message by ending that text with a zero. This is the method which is used by the STROUT routine of the 6502. If you examine 6809 text messages you'll find all of them end with a zero character to mark the end of string text (as we do below). Next, since we have 2 index registers, x and y, we will let one of these registers point to the message and another of these registers point to the location in memory where we want to place the message. The character by character text of the message is sent thru the 'a' register of the computer. The code to accomplish this task is presented below:

```
; send a message to the screen
screen equ $8000 [tell the program where the screen is.]
       ldx # msg [point the x register to the text to send.]
nextc  lda ,x+   [indexed addressing next issue in detail.]
       beq done  [if this is zero the message is complete.]
       sta ,y+   [store the message on the screen.]
       bne nextc [go get another character of the message.]
done   swi       [the task is finished so exit to monitor.]
msg    fcc "This is an example of a string text message. "
       fcc "A long string is sent by the continued use of "
       fcc "the fcc directive of the assembler.  To end this "
       fcc "message the fcb directive is used with a zero which "
       fcc "indicates to this program that the message is "
       fcc "finished. "
       fcb 0
       end [This is the end of the program.]
```

The finished program is ready to be placed on the disk with a: p text.asm command to the EDITOR. Since the starting location of the code remains the same as that of the program ex1.cmd, there is no point in creating a separate cmd file for this program. We need only clear out the text in the microEDITOR and load the cmd file for our first program. This is accomplished with the command *d to the editor to clear text area followed by the g ex1.cmd to load our previously created cmd file. Clearly if we could change all of the "ex1"'s in this program to read: "text" our task of creating the cmd file would be completed. Fortunately we are able to be accomplish this easily with the edit command: *c/ex1/text/. This search-and-replace command changes 'ex1' to 'text'; we needn't re-enter the file. Now that the cmd file has been completed, we save it to disk with the 'p text.cmd' command.

This ends another installment. Good luck in finding the error. Next time we will take a look at the addressing modes of the 6809 processor. -End-

A SHORT, HANDY PEEK

The program below peeks any number of consecutive hex locations you specify, and writes 5 columns to the screen, showing addresses in hex and contents in decimal. See page 60 for a tiny change which makes this program fall flat on its face.

```
10 ! Peekit: Writes hex locations, decimal peeks
20 print chr$(12)
30 input "Enter hex value of start, number of peeks: ", hx$, end$
40 for ii% = 0 to value(end$)-1
50   print hex$(hex(hx$)+ii%);" =";  peek((hex(hx$))+ii%) ;" ",
60 next ii%
```

Most of you, like myself, bought the SPET for its Basic/Fortran/Pascal and figured that, with time, some 6809 programming might also be in order. The icing on the cake, however, is the bundling of APL into this package. A Programming Language (APL), developed by Dr. K. E. Iverson in 1962, is a great language for a micro for a number of reasons. Because of its powerful primitives, the amount of code you will write and interpret will be small relative to the other languages. This means speed! When you invoke an APL primitive, you're pulling in a 6809 routine to do the job: more speed! For example, inverting a data matrix in Fortran is very slow with its interpreter, but APL's "domino" gets the job done in an acceptable time. And you don't have six pages of code either!

Second, the APL processor allows you to extend the language at will. User-built functions conform to APL's simple syntax and can be intermingled with other APL operators. Thus, your command parsing is built-in. APL recognizes your command, finds the routine, and executes it. If you are comfortable with vectors and matrices, you will love APL. As you become familiar with the language, you will find that the SPET is an extremely powerful and flexible "desk calculator". APL's input/output to the terminal is simple and makes data handling easy. In addition, the handling of disk files is straightforward. Finally, APL's workspace concept means that you can keep functions and data together in a very flexible fashion and, if you need something from another workspace, you can simply read it in.

Lest I sound too optimistic, there are a few drawbacks. If you are only comfortable manipulating data element by element (or bit by bit, for that matter), then APL will prove hard to fathom. It requires a different "mind set" and its looping and branching capabilities are not well developed. The different character set often requires translating between ASCII and APL; the lack of lower case letters is often bothersome, and you probably can't print out APL characters on your printer. Finally, once you get "rolling" in APL, you will find that a 28K workspace is just not big enough! The "mind set" is, of course, up to you. We'll cover solutions to the other problems in the coming months.

This column will explore various features of APL and provide lots of coding examples. It is not intended to be a "rehash" of the standard APL introductory material; that exists elsewhere. Instead, the emphasis will be on building tools to get you started in APL and doing useful and/or amusing things quickly. Needless to say, contributions are most welcome. The Waterloo APL manual is very good and should be read carefully. However, it is more of a reference manual than an introductory text. If you find that the examples are simply not enough to get you started, two other APL texts are recommended: (1) "APL: An Interactive Approach" by Gilman and Rose (John Wiley & Sons, 1974) and (2) "APL-STAT" by Ramsey and Musgrave (Lifetime Learning Publications, 1981). Both are available in paperback. All work in this column will be done in Version 1.1 of WCS APL.

Since functions are so important in APL, we immediately turn to the function editor. Once we master the mechanics of using the editor, we will examine some things to keep in mind when designing functions for use in your system. All functions in APL consist of a header that declares the name of the function followed by one or more lines of APL. The form of the header also indicates the

way in which it will be invoked (with or without arguments), how it will return information and whether or not any of the variables within the function are "local". Since the function's header can also be edited, we can change the form of the header later. Nevertheless, the design of the function is of paramount importance since it determines the way your function interacts with other functions in the workspace and with APL primitives. As a first step, we need to become familiar with the editor. Two APL symbols are very important to the editor:

THE FIRST IS 'DEL' OR ∇ (SHIFTED G ON THE ASCII KEYBOARD); THE SECOND, 'DELTA', OR Δ (SHIFTED H ON THE ASCII KEYBOARD).

THE ∇ OPERATOR MOVES YOU FROM IMMEDIATE EXECUTION TO THE FUNCTION EDITOR. IN EXAMPLE 1.1, WE DECLARE THE FUNCTION NAME AS 'TEST1' BY TYPING ∇TEST1 AND HITTING <RETURN>. SINCE NO SUCH FUNCTION EXISTS IN THE WORKSPACE (WS), THE THE EDITOR PROMPTS US FOR THE FIRST LINE OF INPUT WITH: [1], AND LEAVES THE CURSOR ON THAT LINE. TYPE IN: 'THIS IS A TEST' AND HIT <RETURN>. WE ARE THEN PROMPTED FOR ANOTHER LINE: [2]. LEAVE THE EDITOR BY TYPING ∇ ON THIS LINE AND THEN HITTING <RETURN> AGAIN. WE ARE NOW BACK IN IMMEDIATE MODE. HIT THE 'PF3' KEY (SHIFT 3 ON THE NUMERIC PAD) TO SEE EVERYTHING THAT IS DEFINED IN THE WS. WE NOW HAVE A FUNCTION ENTITLED 'TEST1'. TO EXECUTE THIS FUNCTION, TYPE: TEST1 AND HIT <RETURN>. VOILA--THERE'S OUR MESSAGE.

EXAMPLE 1.1:

```
A...∇TEST1
    [  1]   'THIS IS A TEST'
    [  2]   ∇

        TEST1
    THIS IS A TEST
```

```
B...      ∇TEST1[□]
    [  0]    TEST1
    [  1]   'THIS IS A SILLY TEST'
    [  2]   'DON''T YOU AGREE?'
    [  3]   ∇
         TEST1
    THIS IS A SILLY TEST
    DON'T YOU AGREE?
```

```
C...      ∇TEST1[□]
    [  0]    TEST1
    [  1]   'THIS IS A SILLY TEST'
    [  2]   'DON''T YOU AGREE?'
    [  3]   [1.1]'(I WOULD ARGUE)'
    [1.2]   [3]'GOODBYE'
    [  4]   ∇
         TEST1
    THIS IS A SILLY TEST
    (I WOULD ARGUE)
    DON'T YOU AGREE?
    GOODBYE
```

```
D...      ∇TEST1[□]
    [  0]    TEST1
    [  1]   'THIS IS A SILLY TEST'
    [  2]   '(I WOULD ARGUE)'
    [  3]   'DON''T YOU AGREE?'
    [  4]   'GOODBYE'
    [  5]   [Δ4]
    [  4]   '(I WOULD ARGUE)'  EDITED [2]
    [  5]   [Δ2]
            ∇
         TEST1
    THIS IS A SILLY TEST
    DON'T YOU AGREE?
    (I WOULD ARGUE)
```

LET'S EDIT THE FUNCTION AGAIN. TYPE: ∇TEST1 AND HIT <RETURN>. THE FUNCTION ALREADY EXISTS, SO WE ARE NOW PROMPTED FOR THE NEXT LINE, [2]. TO SEE WHAT OUR FUNCTION LOOKS LIKE, CLEAR THE SCREEN <SHIFT CLEAR> AND THEN HIT THE 'PF1' KEY, (SHIFT 1 ON THE NUMERIC PAD). NOTE THAT OUR HEADER IS NOW ON LINE [0]. WE CAN USE THE SPET'S CURSOR CONTROLS TO EDIT ANY LINE ON THE SCREEN. CHANGE LINE [1] TO 'THIS IS A SILLY TEST' BY MOVING THE CURSOR TO THE 'T' IN 'TEST' AND USING THE 'INSERT' KEY TO INTRODUCE SIX SPACES. NOW TYPE 'SILLY', MOVE THE CURSOR TO THE END OF THE LINE AND HIT <RETURN>. THE EDITOR WILL RECOGNIZE THAT LINE [1]

HAS BEEN CHANGED AND WILL AGAIN PROMPT YOU FOR LINE [2]. (*NOTE THAT ANY CHANGES YOU HAVE MADE ON A LINE ARE NOT REGISTERED UNTIL YOU HIT <RETURN> AT THE END OF THE LINE. THUS, IF YOU DECIDE NOT TO CHANGE A LINE AFTER ALL, MOVE THE CURSOR TO ANOTHER LINE*). *LET'S ADD TO THE FUNCTION BY TYPING 'DON'T YOU AGREE?' AFTER THIS PROMPT. WE ARE NOW PROMPTED FOR LINE* [3]. *LEAVE THE EDITOR AGAIN BY TYPING* ∇ *AND <RETURN>. EXECUTE THE FUNCTION AGAIN BY TYPING: TEST*1 *<RETURN> IN IMMEDIATE MODE.*

NOW, LIST THE FUNCTION IN IMMEDIATE MODE BY TYPING: ∇*TEST*1[⎕]∇ *<RETURN>. THERE IS THE LISTING, BUT WE CAN'T EDIT IT: WE'RE STILL IN IMMEDIATE MODE. TO EDIT, ENTER THE SAME LINE BUT TAKE OFF THE SECOND* ∇. *WE GET THE LISTING AGAIN, AND WE ARE LEFT IN THE EDITOR THIS TIME. WHILE WE'RE IN THE EDITOR, LET US INSERT A LINE BETWEEN* [1] *AND* [2]. *WE CAN OVERRIDE THE PROMPT FOR LINE* [3] *BY TYPING THE LINE:* "[1.1] *'I WOULD ARGUE'*" *<RETURN> ON THE SAME LINE. BY USING A FRACTIONAL LINE NUMBER, WE TELL THE EDITOR TO INSERT THE NEW LINE BETWEEN* [1] *AND* [2]. *THE THE EDITOR NOW INCREASES THE LINE NUMBER BY THE NEW INCREMENT*, .1, *RATHER THAN BY THE DEFAULT OF UNITY. THIS IS HANDY IF WE WISH TO INSERT SEVERAL NEW LINES BETWEEN* [1] *AND* [2]. *SINCE WE DON'T WISH TO INSERT SEVERAL LINES, LET'S GO BACK TO THE ORIGINAL NUMBERING SCHEME BY TYPING:* [3] *'GOODBYE' <RETURN> AFTER THE PROMPT OF* '[1.2]'. *THE NEXT PROMPT OF:* [4] *INDICATES THAT WE ARE BACK TO LINE NUMBER INCREMENTS OF* 1. *DID OUR INSERT WORK? TO CHECK, CLEAR THE SCREEN AND HIT 'PF*1'. *THIS RENUMBERS THE LINES AND SHOWS US WHAT THE FUNCTION WWOULD LOOK LIKE IF WE WERE TO NOW LEAVE THE EDITOR. SURE ENOUGH, EVERYTHING IS IN THE RIGHT ORDER. GO BACK TO IMMEDIATE MODE BY TYPING* ∇ *<RETURN> ON THE NEXT LINE. CHECK EXECUTION WITH: TEST*1 *<RETURN> IN IMMEDIATE MODE.*

EDIT THE FUNCTION ONE MORE TIME. TYPE: ∇*TEST*1[⎕] *<RETURN>. THIS GIVES A FRESH LISTING TO WORK ON. WE'RE NOW PROMPTED FOR* [5]. *RATHER THAN ENTER ANOTHER LINE, LET'S DELETE LINE* [4] *BY TYPING:* [∆4] *<RETURN*]. *NOW WE'LL MOVE LINE* [2] *TO THE END OF THE FUNCTION. TO DO THIS, MOVE THE CURSOR UP TO LINE* [2] *AND CHANGE THE LINE NUMBER TO* [4]. *THEN MOVE THE CURSOR TO THE END OF THE LINE, AND HIT <RETURN>. PRESS 'PF*1' *TO SEE WHAT HAPPENED. WHOOPS--WE'VE MOVED THE LINE, BUT IT'S STILL AT* [2] *ALSO. NO PROBLEM--WE JUST HAVE TO REMEMBER TO DELETE THE OLD LINE BY TYPING:* [∆2] *ON A NEW LINE. GOT IT THIS TIME? HIT 'PF*1' *ONE MORE TIME TO BE SURE. THEN LEAVE THE EDITOR WITH* ∇ *AND EXECUTE THE FUNCTION WITH: TEST*1 *<RETURN>. TIRED OF THIS FUNCTION? LET'S REMOVE IT FROM THE WS BY TYPING:*)*ERASE TEST*1 *<RETURN>.*

NEXT TIME, WE'LL TAKE A CLOSER LOOK AT THE FUNCTION'S HEADER AND BEGIN TO CONSIDER HOW TO DESIGN A FUNCTION.

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

If you sit down and work the problems Steve Zeller presents above, you may find yourself receiving mysterious ERROR signals when you're doing everything right. Press PF3, and look at the entries following 'SI'. A lot? Clear them with right-arrow <RETURN> (keep at it until SI is clear). Errors disappear. This had us going up the wall until Steve passed the word.

THE THIRD KEYBOARD We all know SPET has two keyboards: one for the APL font when APL is loaded, and a second for the Waterloo font in the other languages. Well, there's a third. Poke the APL font from any language but APL. Try the board. It's neither Waterloo Roman nor APL. Since we must use the APL font in the microEDITOR to edit and integrate text and programs, we define the third keyboard on the next page. The printouts show the shifted APL

symbols above the symbols of the unshifted keys, row by row. The Third Keyboard symbols apply in all languages but APL when the APL font is on screen.

APL NO SHIFT	-	1	2	3	4	5	6	7	8	9	0	(	+	≥
APL SHIFTED	-	¨	)	<	≤	=	>	]	∨	∧	0	≠	×	$
ASCII keys	_	1	2	3	4	5	6	7	8	9	0	:	-	^

APL NO SHIFT	*Q*	*W*	*E*	*R*	*T*	*Y*	*U*	*I*	*O*	*P*	←	⊢
APL SHIFTED	?	⍵	∈	⍴	~	↑	↓	⍳	○	*	{	⊣
ASCII keys	q	w	e	r	t	y	u	i	o	p	[	\

APL NO SHIFT	*A*	*S*	*D*	*F*	*G*	*H*	*J*	*K*	*L*	[	¯	→
APL SHIFTED	⍺	⌈	⌊	_	∇	∆	∘	'	⎕	⍎	◇	}
ASCII keys	a	s	d	f	g	h	j	k	l	;	@	]

APL NO SHIFT	*Z*	*X*	*C*	*V*	*B*	*N*	*M*	,	.	/		
APL SHIFTED	⊂	⊃	∩	∪	⊥	⊤	|	;	:	\		
ASCII keys	z	x	c	v	b	n	m	,	.	/		

Now, there's a project for Waterloo or someone clever: when you poke the APL font (or get it in the Monitor) and are not in APL, the keyboard should be APL. Ye ed is a touch typist; one mad APL hunt-and-peck keyboard is bad enough; two boards are intolerable. In the time it takes to do one APL page on the 'Third Keyboard', we could write ten pages in ASCII and take the afternoon off fishing. Or is there a way to get the pure APL keyboard somewhere in the system library?

WORD PROCESSING ON SPET

SuperPET is enormously versatile -- more than we suspected--at Word-Processing (WP). Associate Editor Gary Ratliff opened a door to terra incognita in late December, and we can safely say now that we can pull into the microEDITOR (mED) any sequential file in any SPET language, including APL and BASIC 4.0, as well as any WordPro or Wordcraft file. We include files from DEVELOPMENT, but are not sure of the MONITOR. The methods are now fully defined. Text and programs may be integrated in the mED and printed from it in finished form. Say goodbye to the old paste-pot and to hand-typing programs into text. Associate Editor Gary Ratliff is considering a program to pull mED files back into WordPro, but the more he and the editor use the microEDITOR for final printouts, the less inclined we are to go back into WordPro. We find the paging and printing of complex text so simple and so fast from mED that we aren't anxious to pull material back into WordPro.

This issue of the Gazette was integrated (text and programs) in the micro-EDITOR. Part of the issue was written in WordPro; part in Wordcraft; part in the mED directly. All programs were pulled off disk DIRECTLY into the mED--and text wrapped around them there. The editor will never print an issue again by any other means, the method being fast, simple, and totally adaptable to all SPET languages.

This is the first article of a series on WP, starting with the microEDITOR. Those who need an occasional WP system will find it more than adequate. You need learn no new commands, nor lay out several hundred dollars for WP software. Load the mED from the language disk by itself, outside any language. While it will not wrap words to and from the next line when you insert or delete, and will not justify, underline, or print bold face, it has virtues:

(1) What you see on screen is exactly what you will print to printer; (2) marg-

ECLECTIC SYSTEMS CORPORATION
Order TOLL FREE 1-800-527-3135

P.O. Box 1166 • 16260 Midway Road
Addison, TX 75001 • (214) 661-1370

Dramatically Improve Your Programming Productivity

With CCSM© ANSI Standard MUMPS

If you are not familiar with MUMPS you must read the rest of this advertisement

CCSM© is more than just a programming language. It is a well integrated data management system combining with one syntax what other operating systems would call 1) an application programming language; 2) a job control language; 3) a linkage editor; 4) a database management system; and 5) a communications monitor.

PROGRAM MANAGEMENT:
CCSM© provides all programming management facilities needed to manage programs and program files. Programs can be created, edited, cataloged and debugged from within CCSM©. Programs can be as large as disk capacity. A resident algorithm rids memory of least frequently used variables and program modules so that what you need off-disk normally resides in memory.

STRING POWER:
CCSM© makes string handling easy with its extensive set of string operations and functions. Variable length strings can be used routinely without the obstacles presented by most other programming languages.

PATTERN MATCHING:
CCSM© can "filter" user input with a useful pattern matching that will result in fewer user or device errors. For example: dates, zip codes and names can be tested for validity with a single statement.

GLOBALS:
CCSM© obviates the need for traditional read and write operations on secondary storage devices by allowing data elements to be directly referenced as a set of subscripts; all the details of file organization and retrieval are handled by the system.

TIMING:
CCSM© enables a programmer to associate timing constraints with several operations. This feature allows testing for terminal malfunctions as well as prompting users in time-critical dialogue.

DATA BASE MANAGEMENT:
Sorts and merges are not necessary as CCSM© automatically stores data in a dynamically allocated balanced tree structure. Random access to any data item requires at most three disk reads.

CCSM© UNMATCHED IN PROGRAMMING PRODUCTIVITY:
System houses that program in CCSM© (MUMPS) find that their costs are lower than those of their competitors using other languages. Fewer lines of code are necessary per application. Dimension statements are not required. Subscripts may be alpha, numeric or any legal string. Data types need not be defined and can change freely throughout as CCSM© can recognize when it is dealing with alpha, numeric, integer or floating-point data types. CCSM© gives the professional programmer a full set of software tools designed for real-life tasks and problems he consistently encounters in the production and maintenance of application software. CCSM© adheres rigidly to ANSI MUMPS standards, which make it transportable to larger processors manufactured by DEC, TANDOM, DATA GENERAL, HARRIS and others. Additionally CCSM© gives the less-experienced programmer the tools to do a professional job on formidable programming applications.

CCSM© is the Price/Performance Leader!

The most advanced system design for small machines. CCSM© departs from the traditional MUMPS partition concept with state-of-the-art computer software techniques. CCSM© utilizes a complete virtual memory concept to provide the following features:

- No limitation on routine size.
- No limitation on local variable symbol table sizes.
- Only a single copy of any routine resides in memory. (i.e., multiple users take advantage of a single copy of a routine.)
- Only those parts of routines actually being used are memory resident.
- DO's of other routines take no longer than DO's of local labels.

CCSM© is available for the following 6809 systems:

Commodore SuperPet (single-user)	HAZELWOOD Computer Systems HELIX
TANO Outpost-11	GIMIX
Radio Shack Color Computer	Southwest Technical Products

Multi-User systems (up to 16) for $800.00

You may order from ECLECTIC SYSTEMS by calling toll free 1-800-527-3135 from 10AM to 4PM CDT Monday through Friday. Texas residents call 1-214-661-1370.

Or you may write to ECLECTIC SYSTEMS CORPORATION, 16260 Midway Road, Addison, Texas 75001.

ins set easily when you (a) offset the left printer margin to the right, and (b) control the right margin by the number of characters on a line. With 12-pitch type (12 characters per inch), 80 characters per line is fine (a screenful). If you want a wider right margin, or have 10-pitch type or larger, draw a right margin on the screen with a fluorescent felt-tip pen or a clear but visible strip of translucent tape, or a bit of white, fine string. Hit <RETURN> when your text approaches it. Yes, mED saves pure text.

Better, (3) mED needs no obscuring and confusing format commands; (4) it saves your whole text as a continuous page in memory. Paging is a snap; (5) headers and footers (including page numbers) are easily pulled off disk and inserted into text with a 'get'. If, for example, you want 54 lines per page, type: 54 <RETURN> at the command cursor in mED. Screen cursor goes to line 54. 'Get' the footer (or page number). Then, just below the line of the page number, say: +54 <RETURN> at command cursor, and the screen cursor jumps forward 54 lines. You're ready to 'get' a footer, page number, and header at the next page break; (5) last, you can, of course, get ANY material off disk--repititive forms, programs in any SuperPET language, or text created in WordPro or Wordcraft.

If paging manually in the computer age bugs you: we've used IBM, Wang, Lanier, and CP/M machines running WordStar. Despite the clever line-locks, text-ties and conditional forced-paging methods available, all the above are too stupid to page intricate, long text correctly the first time through. Pages always break at the wrong place. We usually make five passes through WordPro's global output to page the normal ten pages of the Gazette (at least 2 hours). In less than half an hour the editor manually paged and printed this intricate issue of over twenty pages. Note we said 'intricate'. Plain text is best paged automatically.

The mED pages superbly not only because it counts and locates lines so well, but also because a change on one page does NOT cascade into all succeeding pages. If you avoid a bad page break (caption at page bottom; text on the next page) by adding an extra line on that page, the remaining pages remain untouched. You are not trapped by WP software insisting that you must have EXACTLY the same number of lines on every page.

Free memory in SPET will hold some 500 lines, single-spaced (about 9.3 pages). If you want more, no problem. Save any sequence of complete pages to disk. Keep the last, partial page as the start of another file. It's a snap with (1) a 'put' to disk using the RELATIVE line commands of mED for the complete pages, and (2) a delete command from from mED to get rid of those pages in machine RAM. You're left with the partial page, the start of another file. Suggest you work with six-page files, to leave room for additions and revisions.

Next issue: How to pull WordPro and Wordcraft files into the mED, BASIC files into mED or into WordPro; use of program PRINTALL for automatic paging, footing, and heading of simple text; handling APL files intermixed with ASCII files.

ON MUMPS You'll note an advertisment on the previous page for MUMPS. We've been curious about it for some time, since it's one of the few commercial pieces of software available for SuperPET. We'd like to run it, but will have to confine ourselves to what we read: that MUMPS is (1) a programming language, (2) an integrated data management system with one syntax for (a) applications programming, (b) job control, (c) linkeage editing, and (d) monitoring any communications with other computers. It was developed at Massachusetts General Hospital as a Utility Multi-Programming System (you can get MUMPS out of that if you try hard). The objective: a simple high level language which will handle the

data base easily and powerfully. No, we did not get the information above from the advertiser, but from sources we think are objective. If any reader has information or hands-on experience, let us hear what you know.

THE SPUG DISK LIBRARY Our Secretary, Paul V. Skipski, has labored for several months to put together the first SPUG disk. It will be available in any disk format (from 2031 through 8050) for $10 U.S. Write Paul Skipski (address on the masthead, last page). Send money only (no disk); specify disk format. Included on the disk: (1) All programs of moment printed in the Gazette to date; (2) SuperPET diagnostics programs, which cover both the lower 32K of memory and the upper 64, as well as the Serial Port; (3) a number of long programs we had no room to publish, in several languages; (4) our text-handling program, PRINTALL, which processes text created or integrated in the mED, both to screen and to printer, and will handle text and programs in all the SPET languages, incuding APL and BASIC 4.0, as well as text created in WordPro and Wordcraft and integrated in the microEDITOR; (5) instructions for using PRINTALL (which will be supplemented next issue); and, (6) a great deal of material on COMAL, plus, as filler, a few BASIC programs and aids which may be handy. Some of the 'filler' probably won't fit into 4040 format, but all the SuperPET material will.

All programs are coded as to language: 'settime-b', for example, is a program to set time$ in mBASIC; 'settime-p', an mPASCAL version. Material for the mED is coded '-e'; COBOL, '-c', etc.

WARNING on the test programs. Some of them require an RS232 port termination if you don't have anything on that port. Paul V. Skipski makes the termination. We have one, and it's well made and works. Paul will make you one, on order, for $10.00 U.S., including shipping.

If you're wondering at the prices: everything we 'make' on disks and RS232 hardware goes into the SPUG bank account to pay for postage and printing on free trial issues which we mail to anyone who asks. (Over 500 sent out so far, at a cost which is NOT negligible. We figure there are over 16,000 SuperPETS in Canada, U.S., and Western Europe, so we have a long way to go. Our membership grows at the rate of 8-10 owners a week; we have members now from Hawaii through West Germany, Norway, England, and Switzerland).

THE EDUCATIONAL LIST Teachers at a number of high schools, colleges, and universities are members of SPUG; some complain of lack of software for teaching. We suspect there might be virtue in having the teachers talk directly to each other, so that they need not individually invent the wheel at each school. If you're a teacher and think an EDUCATIONAL LIST (names, addresses, area of interest), distributed to all educators who use SPET, would be useful and worth the effort, drop us a postcard. We aren't going to tackle this project unless there's demand for it. At the moment, we propose to charge $2.00 for registration, and to mail two lists out each year to all registrants, leaving it up to them to write or call each other, based on the 'area of interest' notes on the list. Educators, let us hear from you!

SOME ADVICE NEEDED AND SOME PASSED One of our members (Barry Bogart, 2405 West 15th Ave., Vancouver, B.C. Canada, V6K 2Z1) has crazy output to printer on the serial port. It seems to happen on second and succeeding lines, about 7 or 10 characters in from the left margin. Characters simply are deleted at this point. When Barry double spaces text, the problem disappears. Anyone with a solution: write Barry and send us a copy.

George Cordahi also has a problem: Using an MX80 F/T P2, he outputs good APL to printer with a filename of 'ieee4'; but if he uses 'ieee4' from other languages, the printer prints characters per PET ASCII. When he uses the filename 'printer' in all but APL, printer outputs ASCII as it should. We suspect his dealer stuck in some hardware which translates SPET's 'ieee4' output from ASCII to PET ASCII, but if so, why does 'ieee4' work for APL? If you can help, write George at the address on page 45, this issue.

When George asked our advice, we told him to use 'ieee4' on APL and 'printer' on everything else, based on Bert Lance's famous line: 'If it ain't broke, don't fix it.'

Barry Bogart writes that he'd like to see those SuperPET owners who use COMPUSERVE identify themselves as SuperPETters and SPUGers; from his contacts with the I.P. Sharp group, he knows of about 50 SuperPET owners using APL, but adds that 'there must be many, many more SuperPET owners on COMPUSERVE.'

CONFUSION AMONGST AND AMIDST EDITORS

SuperPET has three built-in editors, and from letters we see a general confusion in their names. One editor is the APL editor, and runs only in that language. The second is the microEDITOR, which is common to all other languages (with certain variations adapting it to that language). We haven't been consistent in abbreviating its name, so from here on out, it's the mED. Unfortunately, the mBASIC manual calls the mED the 'General Editor', and gives no name to the editor you have at hand in mBASIC as soon as it's loaded--which we christen the 'mBASIC editor'. We'll avoid the term 'General Editor' because it's confusing. To recap: the editor common to all languages but APL is the microEDITOR (mED). In mBASIC, you have two editors, the 'mBASIC editor', which is always available in immediate mode, and the microEDITOR, which you call up with: edit <RETURN>.

NEGATIVES, HEX, AND INTEGER ARITHMETIC

From previous material this issue, you know that you can peek, poke, or address in mBASIC in positive decimal integer, negative decimal integer, or in hex alone. If you use decimal integers above 32767, SPET converts them to negative values. For example, say: i% = 32800. If you then ask: ? i%, it will equal -32736. But SPET will peek or poke the right address, and will not give you an 'overflow' error. Statements such as: poke 32900 + i% will run without trouble so long as i% does not push you over 65536 bytes. Peeks, pokes, and addresses are one matter. Arithmetic operations are quite another, as we see below.

If you write a little for...next loop which requires arithmetic--WOOPS! See program, left. It gives you can immediate 'OVERFLOW' error, but continues to run. SPET warns you that you must beware the results. In this program, everything is okay. SPET peeks the right addresses. But if this were a computation in which you sought numerical results, you'd be in bad shape. You can avoid the error message from the program above by the revised program at left, below. The result is the same, but without an error message (note previous paragraph on the freedom to peek, poke or address without a problem). Now for the next question: what happens when you cross the threshold of 32767 with hex?

```
20 for i% = 32767 to 32767+4*80
30   print peek(i%);" ";
40 next i%    ! Example 1
```

```
10 screen% = 32767
20 for i% = 0 to 4*80
30   print peek(screen% + i%);" ";
40 next i%    ! Example 2
```

Try the little peeks below, using hex, in immediate mode. They work well. Now, knowing that hex peeks work okay, try the same peeks in a for...next loop. (Use program 'peekit', on page 52, this issue). Enter 'peekit'; then change ii% to ii wherever found; tell 'peekit' to start at $7ffa and to iterate 7 times (which should peek $8000 as the last address). WOOOOPS! Find an 'illegal quantity error'? When the loop exceeds 32767, the program refuses to do the job! Why does it run okay when everything is in hex and integers, and refuse to run when you make 'ii' a floating point value? We haven't sorted this one out yet, but we do know that so long as you intermix hex and integers, you can cross the boundary at 32767 without overflow error. Mix hex and floating point and the result is instant disaster. Note we can louse hex up in immediate mode. Ask: ? hex$(34750). You'll get an error. You must ask ? hex$(34750 - 65536). Also note that Version 1.0 rejects -32768 as an integer value. Version 1.1 accepts it, as it should.

```
? peek(hex('7fff'))
? peek(hex('8000'))
```

If you want speed, negative peeks and pokes let you stay within integer limits, but hex peeks and pokes are just as fast, and work easily with integer arithmetic. In mFORTRAN, you must employ negative decimal numbers in peeks and pokes above 32767; the language will not accept peeks or pokes above decimal 32767. It will, however, accept hex peeks and pokes, just as mBASIC will. We discuss mPASCAL later. Since the Systems Overview manual gives addresses in hex, conversion to decimal (either by you or by SPET) just slows things down. When first we mentioned negatives pokes, peeks, and addresses to Associate Editor Terry Peterson, he said: 'Negative pokes are for Apples and the birds.' He's right.

Not only is hex fast, it works in harness with integer arithmetic in mBASIC. If you revise the programs below, written by Terry Peterson, you'll find that no conversion to integers of hex values will speed the programs up. We ran a series of tests of pure integer, pure hex, and of hybrids. Pure hex and hex-integer hybrids run just as fast as pure integers, at least in mBASIC. We'd like to get a specific definition of hex and integer use in mFORTRAN and mPASCAL in the detail we've done it above for mBASIC, if there are any substantial differences. Suspect there are few, and that most limitations emerge from the nature of an 8-bit processor, as implemented in the 6809. Integers are stored in SPET in 2 bytes; if we reserve one bit for the sign, we have 15 bits left; 2 to the 15th power is 32768 ($8000). You can't store that, in binary, in 15 bits:

7	6	5	4	3	2	1	0	Power of 2	
8	7	6	5	4	3	2	1	Bit number	LOW BYTE
128	64	32	16	8	4	2	1	Value of Bit, if Set	
(15)	14	13	12	11	10	9	8	Power of 2	
(16)	15	14	13	12	11	10	9	Bit number	HIGH BYTE
(32768)	16384	8192	4096	2048	1024	512	256	Value of Bit, if Set	

Bob Davis, Associate Editor in mPASCAL, comments that 'there's a significant difference between microBASIC and microPASCAL in handling integers; be aware of it when shifting from one to the other. mPASCAL does not indicate an error on integer underflow and overflow! It just keeps on adding or subtracting bits to the rightmost position (in the registers) and letting them fall off or putting them on the leftmost position as appropriate. Apparently some versions of PASCAL will return a run-time error on integer underflow or overflow; one of the better texts says to program a check of the magnitude of integers to avoid such errors. I recommend such a check with Waterloo PASCAL.'

Bob continues: 'You can peek or poke with positive or negative integers in

mPASCAL indiscriminately. I heartily agree with the manual which says it is more convenient and simpler to use positive addresses.'

We ran a little mPASCAL program which illustrates the trap in mPASCAL. It adds 32768 to 32767, integer-style. The answer comes out: -1 [65535 - 65536]. When overflow occurs, SPET simply converts to the negative integer with no warning, which can raise some *!%$ with results. - End -

Last month, we published a package of procedures to draw, save, and retrieve SPET graphics, both poked and printed, and promised a better scrn_get. Instead, find two programs, which work together; one saves all graphics from SuperPET (from ordinates 1-11 as well as others); the second retrieves all graphics. Substitute them for the programs published last issue, which work only for keypad graphics. Both programs were written by Terry Peterson.

NEW SCRN_SAVE
NEW SCRN_GET

There's a trap in the poked graphics (ordinates 1-11); STOP is chr$(3); when you try to get it off disk, as part of aa$, below, it STOPs recovery of aa$. Terry therefore converts all chr$(3)'s to chr$(14) in scrn_save, below, to cure the problem; then reconverts chr$(14) to (3) in scrn_get before it is poked. Since this process isn't needed for the keypad graphics, Terry offers an alternate recovery and printing process in scrn_get, which prints the keypad graphics very quickly [more specifically, use the 'print' option for any graphics with ordinates above chr$(31)]. Last, Terry recommends you not use ordinates above chr$(11) for poked graphics; the small white square obtained is equally available from chr$(0), which works nicely. The programs below assume you follow this advice. Note that Terry passes parameters to the procedures below; they are 'file$', the name of the file when saved, and 'nlines', the number of lines of graphics to save. Enter these string and numeric values when you call procedures; i.e.:
call scrn_save ('picture', 9), where 'picture' is the filename, and you want '9' lines of graphics saved.

```
 10 proc scrn_save (file$, nlines)
 20 aa$ = ''
 30 for ii%=hex('8000') to
    hex('8000')+nlines*80-1
 40   jj%=peek(ii%)
 50   if jj%=3
 60     aa$ = aa$+chr$(14)
 70   else
 80     aa$=aa$+chr$(jj%)
 90   endif
100 next ii%
110 open #12, file$,output
120 print #12,aa$ : close #12
130 endproc
```

Scrn_get, below, works in exactly the the same way. You must enter the filename in parens when you make a call to scrn_get. You can easily modify the procedure to pick either the 'print' or 'poke' options; the 'print' option is by far the fastest, but works only if no ordinates < chr$(32) are in the disk file.

```
 10 proc scrn_get (file$)
 20 open #11, file$, input
 30 linput #11, aa$
 40 ! the next single line substitutes
 50 ! for the rest of procedure if no
 60 ! ordinates < chr$(32) in image.
 70 pp%=cursor(0):print chr$(1):pp%=cursor pp%
 80 !
 90 while idx(aa$, chr$(14))
100   ii% = idx(aa$,chr$(14))
110   aa$(ii%:ii%=chr$(3)
120 endloop
130 jj%=1 : mm%=jj%
140 for ii% = hex('8000') to
    hex('8000')+len(aa$)-mm%
```

One other point deserves comment: note how Terry uses pp% = cursor(0). If you go to immediate mode, and put cursor on line 1, left margin (home position) and enter 'pp% = cursor(0)', you'll find on printing 'pp%' that it equals 81. The method stores cursor position. Terry em-

```
        (cont. from last page)
150     poke ii%,ord(aa$(jj%:jj%))
160     jj% = jj% + mm%
170 next ii%
180 open #12, file$, output
190 print #12, aa$ : close # 12
200 endproc
```

ploys it to position cursor (line 70, scrn_get) to about where it was when he issued command to call scrn_get; i.e., to get 'READY' out of the retrieve-image. (0) in the command is a dummy argument; you can as easily use (a) or (whatsit). If variable names such as (a) or (whatsit) have value, however, the command will move the cursor to the appropriate screen position; e.g., if 'a' = 1560, cursor will go to 1560. Using '0', as Terry does, makes more sense. (Thanks to Frank Brewster for the same comment.) The method is most useful.

THE SIEVE OF ERATOSTHENES (REVISITED) VISITED The Gilbreaths, in the January 1983 issue of BYTE, published ratings of a host of computers, using the 'sieve' as a benchmark. Being curious, we tried the 'sieve' in mBASIC and mPASCAL, faithfully adhering to the spirit of the benchmark. We were startled, for mBASIC ran a lot faster than we thought it would, but mPASCAL was incredibly slow. We couldn't benchmark APL, because there is not sufficient memory in SPET to handle an array of 8190 floating-point numbers, and APL in SPET won't handle integers. Even so, Steve Zeller ran the sieve up to a value of 1000; so we ran the mBASIC program for the same value (below). mAPL is rather slow, too. Both mPASCAL and mAPL are interpreted in SPET; if they were compiled, the difference in run times would be large (but so would it be for a compiled mBASIC). We'll have more to say next issue on what we learned using ten different mBASIC programs for the 'sieve'. One startling fact: if you're using integer arithmetic in a for-next loop, even converting '1' and '0' to integer values saves significant time in a loop which is iterated many times. Here are the run times of the test, and some values from the Gilbreath report:

Language	Computer	Time for One Iteration (seconds : minutes)	
mBASIC	SuperPET	258	4.3 (full sieve, n=8190)
mPASCAL	SuperPET	2060	34.33 (full sieve)
BASIC	PET	318	5.3 (full sieve. Gilbreath)
BASIC	HP 85	308.4	5.14 (full sieve. Gilbreath)
mAPL	SuperPET	92.85	1.55 (n = 1000)
mBASIC	SuperPET	38	0.63 (n = 1000)
BASIC	6502, SuperPET	'under 200'	Report by Terry Peterson
mFORTRAN	SuperPET	94	1.57 (n = 500)

Terry Peterson ran the mFORTRAN trial and said he hadn't time to wait for a larger value of n. Even so, mFORTRAN is slower than mAPL in SPET. Note that we do not know what program the Gilbreath report used for BASIC; Terry notes that he remained faithful to the benchmark, but used a for...next construct instead of what was essentially a 'while' loop. We learned that 'while' loops are slow indeed in SPET, compared to for...next loops. When we changed one, single, inner loop from a while...endloop to a for...next loop, we cut run time from 336 to 258 seconds. Terry suggested it, and it worked beautifully.

Let the times above be no derogation of SPET. The interpretation of mPASCAL and mAPL is a virtue; Jim Strasma, in the latest MIDNITE REVIEW/PAPER, compares UCSD PASCAL with mPASCAL in the classroom, and finds mPASCAL far superior because students can debug with the interpreter, without first compiling (long waits), only to find errors in the compiled code. Moreoever, SuperPET was designed to

operate with a host computer. Debugged code, up and running, can then be compiled for runs on a big, fast machine.

While we deeply appreciate the convenience of interpreted code and the debuggers Waterloo built into the SuperPET software, we'd still like to see compilers made available for SuperPET. The editor uses his in business, and bought it for its structured languages, which make programming a delight. Not very much software is written to handle the financial end of legal work, so we had to write ours. It'd be a delight to be able to compile it, once it's up and running. SuperPET is not solely for classroom use, Waterloo. The commonest question we are asked is: "When do we get compilers?" The sooner the better. About 10 per cent of our members are educators. The rest of us want and need compilers. (And so do some of the teachers, from letters we've received.)

SETTING TABS IN ONE INCREMENT

The program below, if called with the increment in which tabs are to be set, will set them evenly across the screen, from an increment of 1 between tabstops on up to 79.

```
50400 ! Set tabs in increments: 'tabeven'
50410 proc tabeven(incr)
50420 for ii = 1 to 17 step 2
50430   poke 270 + ii, kk
50440   if kk+incr >= 80 then flag = 1
50450   while flag and ii<17
50460     ii = ii + 2
50480     poke 270 + ii,0
50490   endloop
50500   kk = kk + incr
```

We don't like the default tabset in SPET, because there is no tabstop at the right margin.

If you edit much, you must be able to tab to the end of any

```
50510   if flag then quit
50520 next ii
50530 poke 288,0,79 ! Tab at right margin
50540 kk=0:flag=0
50550 endproc
```

line, for there you hyphenate and in writing a program place & for continued lines. Tabeven always sets a right margin tab-stop, whatever the increment. If you'd prefer not to have it, change the for-next loop value of "17" and all other "17's" to "19", and strike line 50530. The modified program, with an increment of 8, will then give you a standard Waterloo format. In whatever settings, tabstops not used are set to 0.

This program, together with a settime setdate pair, we keep in a little 'bootup' package, which we run right after loading mBASIC. It sets time, date, and tabs, first thing in the morning. Mighty handy.

A MACHINE LANGUAGE DUMP FROM SCREEN TO PRINTER OR DISK

Jeff Larson, Route 1, Box 261D, Rustburg, VA 24588, runs a big DEC during business hours and a SuperPET at home; the editor on the DEC is a line editor only; Jeff brings his stuff home as an ASCII file and edits in the mED, which he says is far superior. One trouble: he had to copy SPET monitor dumps by hand, and after a few sessions wrote a dump to printer, which we got about two weeks ago, and forwarded to Gary Ratliff. Shortly after, we heard the sound of jubilee from Mississippi; Gary called and wanted to stuff the dump into this issue. We objected; we'd entered and run it; it always dumped twenty lines, even if 19 of them were blank. Blank lines smash the printhead back and forth so violently we threw a ribbon cartridge off the carriage and ground it to bits. We refused to inflict that catastrophe on anybody else. Gary said he'd fix it, and fix it he did, just in time to print. Note that the entry of two "qq's" at left margin will kill the dump. You can print alphanumerics only, not blank lines. (We wish KEYPRINT, the machine-language dump in BASIC, were similarly written.)

Best of all, the program can be modified to dump to disk as well as to printer. We'll try to publish some modifications next issue.

6809 CODE SCREEN DUMP by Gary Ratliff

Jeff Larson brings us his routine to print the contents of the screen. He uses a serial printer. By changing the type and size equates in this program, however, you may have it print to whatever type of printer you wish (see 1st program line). Size 20 is designed to translate listings from the monitor, while an increase of size to 40 will give screen dumps of the monitor. Finally, size of 80 will print all of a screen line. Thanks to Jeff, we'll publish next issue a memory map of the 6809 side of SuperPET.

Old timers may remember purchasing those early disassembly listings for early PETs at $29.95, because Commodore was required by MicroSoft to protect the contents of the interpreter. A lot of the 1979 programs for the PET were designed to tell users how to overcome this and how to get at the contents of BASIC code. My first published article was an 18-byte routine to do exactly that. Waterloo is to be commended for making the discovery of the inner workings of SPET easy for the user. Can you imagine a screen dump routine in 6502 in early '79? No! First we had to dig into the guts and figure out how it all worked. Oops! And another biggie: there were no printers then (at least not from Commodore!).

```
;screen dump routine          dump.asm
xref initstd_
xdef outptr
xref openf_
```

[Ed. This came in one day before we had to print this issue; on trial, it clobbered mBASIC when located at $7000. So we moved it

```
 xref closef_
 xref fputchar_
 xdef prtline

 type equ 2        ; use 2 for printer 3 for ieee4
 size equ 80       ; use 40 for monitor 80 for line

 lds #$0fff        ;Initialize S pointer
 jsr initstd_      ;Initialize standard IO
 ldd # mode        ;Load address of file mode
 pshs d            ;Push file mode into S
 ifeq (type - 1)   ;serial
   ldd # typ1      ;load address of 'serial'
 endc
 ifeq (type -2)    ;printer
   ldd # typ2      ;load address of 'printer'
 endc
 ifeq (type -3)    ;ieee4
   ldd # typ3      ;load address of 'ieee4'
 endc
 jsr openf_        ;Open file
 leas 2,s          ;Remove file mode from S
 std outptr        ;File control block address
 if ne             ;If file opened ok,
   ldd #$8000      ;Load beginning of screen ram
   std line        ;Store in 'line'
   loop
     ldx line      ;Load beginning of line
     jsr prtline   ;Print line (20 characters)
     ldx line      ;Put beginning of line into x
     ldb #80       ;80 is number of columns
     abx           ;Add 80 to get beginning of new line
     stx line      ;Store in 'line'
     ldd ,x
     cmpd #$7171   ;test for 'qq' at start of line
     quif eq       ;printing complete if reached
     cmpx #$8780   ;Compare with screen bottom ($8750 would skip last line)
   until eq        ;Loop until last line done
   ldd outptr      ;Load file control block
   jsr closef_     ;Close file
 endif
 swi
 prtline ldy # size
 loop
   ldb ,x+         ;Put character into b
   pshs y          ;Push number of interations onto S
   pshs x          ;Save x
   pshs d          ;Push d onto S
   ldd outptr      ;Load file control block
   jsr fputchar_   ;Send character to selected device
   leas 2,s        ;Remove file mode from S
   puls x          ;Restore x
   puls y          ;Restore y
   leay -1,y       ;Decrement y
 until eq          ;Quit when y = 0
 ldb #$0d          ;Load carriage return character
```

up to $756e; it still clobbered mBASIC. Then, using a part of Jeff Larson's memory map, we put a top-of-memory pointer in at $756c--and it works like a charm. Find below the specific instructions on how to enter the dump, load it, and use it. (And for that pointer!)

Entering: Load DEVELOPMENT, and enter the mED. Create dump.asm (first program left) and file it. Then enter dump.cmd (second program left, below), and file that also. Put both files on disk in drive 0. Put a language disk in drive 1 (for exports). Then assemble and link (See Gary Ratliff, Vol. 1, pp. 33-36) on how. Then, in the monitor; say:

>l dump.mod <RETURN>

Then get your printer ready, because when you give 'g', below, you'll dump the screen. Suggest you enter a 'qq' on a blank line ABOVE repeat ABOVE the next command to stop the dump:

>g 756e <RETURN>

And stand back!

If you intend to work with any high-level language in SuperPET after 'dump' is in memory, you must change the top-of-memory pointer. If you call 'dump' with a 'sys' call without doing this, SPET crashes. Thud.

Again, thanks to Jeff Larson, we know how to do that, too. The top of memory pointer is found in two bytes, $0022 & $0023. AFTER repeat AFTER 'dump' is in memory, poke as follows: (find hex and decimal versions below)

```
pshs d          ;Push onto S
ldd outptr      ;Load file control block
jsr fputchar_   ;Send to device
puls d          ;Restore d
rts
```

```
typ1    fcc "serial"
        fcb 0
typ2    fcc "printer"
        fcb 0
typ3    fcc "ieee4"
        fcb 0
mode    fcc "w"
        fcb 0
outptr  rmb 2
line    rmb 2
end
```

And here is the dump.cmd file for the linker:

```
"dump"
org $756e
include "disk/1.watlib.exp"
"dump.b09"
```

```
poke hex('22'),hex('75'),
                hex('6c')
                    OR:
poke 34, 117, 108
```

In mBASIC, top of memory is at $7fff; you can confirm top of user memory with peeks of $22 and $23; but we were in a rush and thus backed off more than the 86 bytes needed to hold the code for 'dump'. The top of memory location can be written into this program; it may change with language (no chance to check). Suggest you confirm top of memory before using 'dump' in other languages. When you peek $22 and $23, convert the high byte ($22) and the low byte ($23) to hex, and join them in order. The values 117, 108 (above), thus converted, become $756c.

The entire 'dump' DEVELOPMENT file is on the SPUG disk announced this issue. You can amend any of the files to conform to your language and your printer. Or you can load the module with: >l dump.mod <RETURN> from the Monitor in the microEDITOR in any language which uses it (all but APL). After you've poked the proper values for top of user memory, you can get 'dump' by a SYS call: sys hex('756e') or, in decimal, sys 30062. Bloody well better have your printer ready! We emphasize again: the 'qq' (no quotes) to stop the dump MUST be on a line above the call, whether from language or from the Monitor.

While coding and entering 'dump' may look complicated, we got it into memory the first time we tried, easily; we used the step-by-step instructions Gary Ratliff provided last issue. You enter the program for dump exactly and precisely in the same way as you entered the program to print a simple 'a' at the top left of the screen. Gary laid a firm foundation on method.

A THICK ISSUE This issue is large because (1) the readers contributed, (2) we secured our first advertisers; and, (3) an anonymous member with a large heart gave us the extra cost of printing and postage. Keep the material flowing in! This issue just about dries up the well.

****

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

APPLICATION FOR MEMBERSHIP, SUPERPET USER'S GROUP

Name:_____ Disk Drive: _____ Printer:_____

Address:_____

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

Enclose Annual Dues of $10:00 (U.S.) by check or money order, made out to Secretary, SPUG. Overseas dues: $20.00 U.S. Mail to: Paul V. Skipski, Secretary, SuperPET User's Group, 4782 Boston Post Road, Pelham, N.Y. 10803, USA.

Newsletter published by the SuperPET Users' Group (SPUG): editorial offices at PO Box 411, Hatteras, N.C. 27943. Secretary, Paul V. Skipski, 4782 Boston Post Road, Pelham, N.Y. 10803. Membership applications and inquiries to Mr. Skipski. Newslette material to Hatteras, attn: Dick Barnes, Editor. SuperPET is a trademark of Commodore Business Machines, Inc. Enclose a self-addressed, postpaid envelope with all material submitted and all inquiries requiring reply. Membership: $10.00 per year, U.S. in North America, $20.00 overseas and elsewhere. See enclosed application.

For all outside the U.S.: All nations members of the Postal Union offer certificates good in the postage of any other country for a small charge. The Union includes Canada, U.S., most European nations, Russia, China, and most of Araby. Each Gazette issue weighs one ounce: 20 cents U.S. & Canada, 80 cents to Europe. For other rates, see your local post office.

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

PRINTED MATTER