SUPERPET GAZETTE

The shape of Commodore's future begins to appear. We recently received word that the 8050 and 8250 drives will no longer be produced; this is the end of the PET computer line. We also learned that Commodore Canada is now selling a PC clone which obviously is one of Commodore's new business machines. The other entry is the Commodore 900, which runs a Unix-like operating system, Coherent. Commodore has hedged its bets by matching the business machines of both IBM and AT&T. Last quarter, Commodore lost over $20 million; the C64 isn't selling well; the C128 and the new machines haven't taken up the slack.

Will the new machines be sold in the U.S.? We don't know, for Commodore no longer has a U.S. dealer network. The prices for both new machines are in the range of Radio Shack's PC clones and Unix stuff, which is well supported. If Radio Shack isn't doing too well against IBM ($400 million vs. $2.5 billion in microcomputer sales last year), we think Commodore will do far worse--and is in fact dead in the U.S. business market.

If you plan to eventually replace SuperPET with another computer, what are your choices? Loyalty to a particular brand of computer is nonsense; reasonable folks will buy gear (1) offering gobs of software (2) locally supported for training and maintenance, and (3) with reasonable performance--in that order. There are three alternatives: the AT&T Unix machine, a PC or clone, or a Macintosh. Unfortunately, all options are grievously flawed: the AT&T machine by the complexity of Unix, the lack of commercial software, and the high cost of the system; the PC by MS DOS, which is an archaic, clumsy, complex operating system best suited to computer professionals--not users. We have seen too many PCs and clones sitting on desks as status symbols, gathering dust, to have any illusion that the average business user or secretary can or will cope with MS DOS.

We think the Macintosh as now configured will fail. Its disk accesses are mortally slow; those silly 3.5-inch disks don't hold enough data; you must have two cutesy wee drives for serious work; even then you must love disk-switching. It needs cursor controls on the keyboard (we refuse to tear down a wall to get room on our desk for a mouse) and we aren't about to get a case of "mouse elbow" by reaching cross-country five thousand times a day to move the cursor. The Mac should have been both modifiable and expandable. If you want a hard disk in Mac, for example, you pay $2500 to get the disk, to rip out the 68000, to install the disk, and to relocate the chip (you can stuff a hard disk in a PC for well under $1000). Although Mac is easy to use the day it arrives, you are forever after bound to the slow drag-an-icon, pull-down-a-menu, click-the-mouse drill. It's simple to learn but you'll never speed it up, no matter how experienced you are. As for modifying its routines and writing your own assembly language programs--forget it unless you're a pro. Apple will then sell you the system data and you get to spend six months figuring it out. How is Mac selling? Badly. Steve Jobs was removed as head of the Mac division a couple of weeks back...

Unless Apple redesigns Mac, we think the firm is down the tubes. The majority of its income still comes from the ancient Apple II series, which isn't going to sell forever. Will Apple create a flexible Mac you can use in your own way, an adaptable Mac with open architecture (which made Apple rich), an adult Mac with the disk storage and memory capacity required for business? Mortal question.

Which, all told, is not a happy picture. Upgrade to a new computer? To what? At this writing, Commodore's Amiga Lorraine and Atari's ST series, both based on

the MC 68000, are still in the wings. We're inclined to watch them and to follow the progress of GEM, Digital Research's new Mac-like overlay for MS DOS. If you are new to MS DOS, GEM lets you run a PC as you'd run a Macintosh. As you gain experience, you can leave GEM and go directly to MS DOS. But--we hear that GEM is buggy; if the software houses don't write their stuff to use it... Is there any hope for those who simply want to use a computer without becoming a professional, for those those business people who can't afford to spend a couple of months training a new computer operator every 21 months (which is normal turnover time for such employees in these parts)? Sigh. The computer we'd buy for ourselves is hardly the computer we'd buy for employees Ginnie or Sam, who view computers as tools, and have no interest in them apart from the job. For them, there still isn't a sensible business machine on the market.

DEAR RED: PLEASE RENEW!

If your address label is redmarked, check the RENEW block on the last page and send it with your address label or a copy. Bless us with a renewal check; your ISPUG membership has expired. Also send articles or gifts up to $10 million.

BEDIT'S HOST and UTILITY DISK II Last issue, we warned that BEDIT, ISPUG's new superEditor, wouldn't support HOST and asked those who wanted the capability to wait. Joe Bostic has fixed the problems; the current versions of BEDIT (and BEDCALC) support HOST fully. On some early disks, we furnished a version of DEVCALC (BEDCALC revised for the editor in Development) which wasn't final, and told those who got the disk to send the disk back, upon announcement, to get the final DEVCALC. If you aren't sure which version you have, check the instructions for DEVCALC; if a NOT search is done by: \searchphrase/, you have the early version. Send your disk; we'll give you the final version [in the final, a NOT search is done by: \searchphrase\ ].

Lacking room last issue, we did not show all of the files on Utility Disk II; we supplement the list in the last issue with that below:

71	"DEVCALC"	PRG	A special form of BEDCALC for use in Development
15	"devcalc_instr:e"	SEQ	(with the assembler/linker)
8	"mfor_patch3:bp"	SEQ	The last patch for mFORTRAN issued by Waterloo
7	"mbasic_patch3:bp"	SEQ	Ditto for microBASIC.
9	"notes_patches:e"	SEQ	Instructions on how to use them.
66	"memap_titles:e"	SEQ	SPET memory map sorted by title of routine.
66	"memap_address:e"	SEQ	Same map, but sorted by address. 7 pages each.
111	"bigcalc:be"	SEQ	A program in microBASIC which handles numbers
29	"bigcalc_doc:e"	SEQ	up to 400 digits long. From TPUG.

Utility Disk II may be obtained from ISPUG, PO Box 411, Hatteras, N.C. 27943, for $10 U.S. in 8050 or $16 U.S. on two 4040s. Please state the format!

ONCE OVER LIGHTLY
Miscellany

We thank all those who responded with Gazette articles to our plea for help in Issue 3, Vol. II; this issue carries most of them. Note the diverse interests and skills of the writers; they cover more ground than ever we can, unassisted. But--we're plumb out! Send more or next issue will be thin, thin, thin...

TO REPEAT A SEARCH... One of our gurus (we won't say which) threatened to revolt when he heard that the command to repeat a search string in mED (which is two slashbars //) was forbidden in BEDIT. We asked why he didn't use +/ or -/ to repeat a search forward and backward in text. He was astounded to find out that the commands existed and worked, and promised to burn his guerilla suit.

Why was // forbidden? Well, please tell us at 11 p.m. the last search string you entered at 11 a.m., and why // as a replace string means null, and as a search string means something not null. Should you be tempted to make // mean null, how do you define the difference between NOTHING (the absence of anything), ZERO (a quantifiable absence of something), and NULL (the first and zero character in the ASCII series)? Which does // mean? And, if you then assign any of the meanings above to //, please write the code for the change at left, which asks that all nulls be changed to an X. In any line, how many nulls do we have, and how many X's should be the result? It boggles the mind.

```
c*//X
```

A QUESTION OF NOT AND OR Grab your pencil and paper; jot down the answers to the questions at left, below, when the internal value in register2 is 204 and the integer variable d32%=32.

```
variable1%=peek(register2) and not d32%
```
Q1: What is the value of variable1%?

```
variable2%=204 and not d32%
```
Q2: What is the value of variable2%?

We convert the values of 12 and 32 to binary below to save your time:

Decimal 204=%1100 1100
Decimal 32=%0010 0000

We trust you won't peek at the answers below until you write down your own. Then answer this question: In what order are the AND and NOT performed in the two examples above? Remember that all integer logical comparisons operate bitwise on the operands, and that non-integer operations are logical ones which do not affect the bits.

The answer to Question 1 is 204. The PEEK is returned as an integer; the NOT is executed first; the entire operation is performed on the bits, as at left. NOT complements all bits in decimal 32 (reverses each one). The code is actually executed as if written thus:

```
variable1%=(integer 204) and (not d32%)
```

```
    D204=%1100 1100
 NOT D32=%1101 1111
          ----------
After AND: 1100 1100
```

The answer to Question 2 is 1--or TRUE. The value of 204 is not an integer value; the question is one of relations and says, "set the value of variable2% to 1 if the following statement is true, or to zero if it is not." And the statement asks if both 204 and NOT d32% have value! NOT d32% is a complement of 32, or -33 (it exists); so does 204. You receive, inevitably, a TRUE response. Terry Peterson wrote on this subject in Vol. I, p. 116, Issue 9. Beware!

LET'S MAKE THINGS COMPLICATED We are continually astounded by the words employed by computer priests to impress the laity. Nobody in the priesthood can possibly read a manual or a handbook; it's always documentation. Gee, in the good old days, you documented a case at international law, or the cause of a war. Any day now we expect to see a washing machine sold "with full documentation." This is not an isolated example; consider the phrase "communicating by a null modem". Well, "null modem" means "no modem," or, in short, that two computers talk by direct cable. In an article this issue, Associate Editor John Frost uses "null modem" because the phrase is common; we bow to usage but don't have to like it--any more than we like "paradigm." This supergoo means "model", a word forbidden by the computer priesthood because any peasant can understand it.

SUPERPET SYSTEM FOR SALE Frank Brewster, 1 North Vista Ave., Bradford, PA 16701, 814 368-6319 offers his SuperPET, an 8050 drive, an IDS 445G high-resolution printer/plotter, a Hayes 1200 bps Smartmodem, a Unigard spike protector, an ADA 1450 IEEE-Serial printer interface, plus all manuals, extra paper, cables and disks, for $1500--all equipment in operating condition. He'll deliver within 500 miles of Bradford. Call or write.

REPORT ON OS-9 Russ McMillan, whose SPET wouldn't operate after he installed OS9 gear, reports that his machine now runs okay. TPUG sent schematic data; Russ's dealer found a bad chip on the old board. Richard Jones of Texas, who has an old 3-boarder, tells us "I installed the memory management unit myself, desoldering chip U9 and soldering a 20-pin socket in its place. I have extensive soldering experience. You must take care in soldering the 20-pin socket so that it makes complete connections on top of the PC board as well as on the bottom. This means raising the socket slightly on the board so the soldering iron can get under the side of the socket and solder the top connections. The cable connection fit well even with a raised socket."

Richard also reports that while OS9 tests recommended by TPUG ran okay, his SPET locked up when he loaded the OS9 software and got the copyright display. After a cold start, SPET operated normally. He suspects a bad disk, which has been sent back to TPUG. We've had no other reports of OS9 problems. Nick Solimene of Woodhaven, N.Y. reports his OS9 system on a two-boarder is working fine, though he'd like two or more REL files on an 8050 disk for OS9 files and more details on memory access under OS9. When you read these reports, remember that version 0.9 of OS9 for SPET is a test version, preliminary to release of V1.0. Bugs are to be expected; final manuals and data should accompany V1.0.

FILES WON'T INPUT??? From guru Terry Peterson comes the following note: "I recently figured out the origin of a mysterious problem encountered in trying to input from "ieeeX" files. Sometimes when SPET is turned on in 6809 mode cold, said files would refuse to input--giving instead a phony EOF error. By peeking the FCB's (File Control Blocks), I discovered that SPET maintains EOF status in bit 0 of byte $15 of a FCB, and doesn't clear it for ieeeX files! I show (see at left. Ed.) an easy solution from language. You clear the FCB EOF bit with a phony open/close (or reset) before you open the real file. You have a second option: determine the FCB address by peeking, using mBASIC's pointer to the current file at $58. Then, prior to each input, you 'poke FCB+20,0'. This method doesn't require one to close and then reopen a file, but the peeky-poky is arcane. Last, you can accomplish the same thing by a switch to 6502 and back, which fills memory with $AA (binary 1010 1010); it clears the bit."

```
open #15, "disk",inout
reset
open #15, "ieee5", inout
```

NEW PASCAL COMPILER? From Roger Bassaber, a French member of ISPUG who lives on a island down in the Indian Ocean, we received a disk copy of a compiler for Pascal and sent it out for test. A preliminary report from Russ McMillan says it is pretty good (it runs on the 6502 side), accepting ASCII files generated in 6809 mode; Russ ran all 24 of the Waterloo mPascal examples in the manual, and they all worked with a few minor differences in input/output behavior. But--we found out the compiler is copyrighted (tilt!) and we don't have instructions. We'll continue tests; if they pan out, if we can get permission from the copyright owner and a manual, we'll make it available. Thanks for the tests, Russ.

TO THE GUILLOTINE DEPT. On ISPUG Utility Disk II is program DOS, written by Alain Proulx, which offers an option to print either to serial or ieee4 printers. We got a report that the options are reversed on the menu; pick "serial" and you send to "ieee4" and vice versa. Once you know, it's easy to handle the error. We've fixed the master disk for future releases. Anybody who can't cope with that can send the disk back; we'll put on the corrected version.

Seems that we called the Ted Edwards compiler for APL the "Williams Compiler" in our announcement of the disk on p. 100 last issue. Please change your copy; we aplogize to Ted as we clamber into the tumbril alongside Alain....

EVERYTHING YOU WANTED TO KNOW DEPT. "The Complete Commodore Inner Space Anthology," is a new book available from The Transactor, 500 Steeles Ave., Milton, Ontario, Canada L9T 3P7. The book is a compendium of all the data which has appeared on most Commodore products plus much new stuff. Of interest to SuperPETters are the complete memory maps for 4040 and 8050 disk drives. Maps for the 2031 are missing, though the 2031 is supposed to be similar to the 1541 except for IEEE functions. Data are presented as tables; they include system constants, a RAM memory map with zero page contents at power-up, RAM memory, ROM memory, controller RAM usage and a disk controller ROM map. Included are specifications, directory file header format and sector format, BAM formats, disk data file formats, and error diagnostics. Sorry, we don't know the price. We thank Reg Beck for the information, but are saddened to announce it in the issue which reports the end of production for the 8050 and 8250 (the 4040 died a few months ago).

THAT BUG IS ALIVE DEPT. Save with Replace (SAVE@) is proven buggy on the 4040 and 1541. It has been rumored for years, but was demonstrated recently. An article in the July, 1985 Transactor contains a program which, if run on the 4040 or 1541, results in improperly replaced files. This didn't occur on trials with the 8050 or 8250; the 2031 wasn't tested. The SAVE@ problem is of no concern to those who use the Waterloo languages, but has been in controversy so long that Reg Beck, who sent the item in, thought you'd be interested.

WHICH FORMAT DOES WHAT WITH WHICH TO WHOM The terminology used to define the number conversion routines in the Development manual is confusing. We find undefined references to "integers" and "binary" and do not know whether the terms are interchangeable or define different ranges of value. In this article, we'll try to sort out the number formats and the conversion routines. We confine ourselves to integers; numbers containing decimal fractions are a separate subject.

We begin by discarding the term "binary format"; all numbers, in whatever notation, are stored internally in that format. Second, let us distinguish between notation (the form in which we state a number, usually hex, decimal, or binary) and the actual value of the number itself, which always remains the same, whatever notation we use to express it.

You may view numbers in four forms: 1) as external strings ["7FD0" in hex notation, "32720" in decimal notation, or "0111111111010000" in binary notation], as 2) internal string values ["7FD0" shows in the monitor as $37 $46 $44 $30]; decimal and binary strings appear in their ASCII equivalents; as 3) the binary form in which the string values are stored in memory, and, 4) as the value of the number itself. Decimal ten, for example, is found in the following four forms: 1) external string: 10; 2) internal ASCII: $31 $30; 3) internal binary string:

0011 0001 0011 0000; and, 4) as the value it represents, stored internally in binary notation [1010], which will show in the monitor in hex notation as: a.

Because each number has four possible forms in three different notations, it is difficult to say anything about conversion until you define your terms. We'll essay several, the most important of which is the "counting" or cardinal number.

The Counting Numbers These start with 0 and range thereafter to (and, if you believe Cantor, through) infinity. By definition, they are integers; 1 and 5000 are counting numbers expressed in decimal string notation. If you use "i%" in a a loop, the value of "i%" at any time is a counting number. Waterloo seems to call the counting numbers "integers" when they fall in the range of -32768 to +32767, and sometimes calls them "binary" in the range of -65536 to +65535.

When you view a counting number in memory, in the monitor, it is reported in hexadecimal notation. The counting number 10000 (in decimal notation), for example, is reported as 2710 (in hex notation) in the monitor. Its counting number value is independent of both notations, though we must use one to express it. In all computers we know about, all numbers are stored internally in binary notation. Let us be very clear on the differences between numbers in any notation in the form of a string and that same number in the form of a counting number:

Hex String Value: (on screen)	String Internal Value: (viewed in monitor)	Counting Number: (viewed in monitor)	Binary Internal: (counting number)
"9FFF"	$39 $46 $46 $46	9f ff	1001 1111 1111 1111

A number in string form, in any notation (binary, hex, decimal) must be converted to a counting number if you propose to manipulate it mathematically.

If, at this point, you're thoroughly confused, good! So were we; we looked for and found a straightforward way to handle conversions. Any number, in any string form, in any notation, may be converted to a counting number; similarly, any counting number may be converted to any notation in string format. We define the steps below, and show which system routine does what, with which, to whom:

The Six Basic Conversions

1. Convert String Notation (on screen) to Counting Number (Internal):

   a. Decimal notation to a counting number: System routines DECIMAL_ and STOI_.
   b. Hex notation to a counting number: System routines HSTOB_ and HEX_.
   c. Binary notation to a counting number; No system routines. See below.

2. Convert Counting Number (Internal) to String Notation (on Screen):

   a. Counting number to decimal notation: System routine ITOS_.
   b. Counting number to hexadecimal notation: System routines ITOHS_ and BTOHS_.
   c. Counting number to binary notation: No system routines. See below.

On the following pages, we summarize data about system conversion routines.

PRINTF_. If you use the %d option in it (to incorporate in the string printed a variable value in decimal notation), PRINTF_ will convert a counting number to a decimal string. Example: If a loop has executed 15 times, the monitor will show a hex value of OF for the variable. PRINTF_ will print the value as decimal 15. It automatically converts any counting number to decimal notation--so long as you do not exceed 32767. Beyond this point, the routine fails.

Likewise, the %h option in PRINTF_ will print a counting number in hex notation. It is unfortunately limited to the value of 255. PRINTF_ makes serious errors if you give it string format (in which a zero is represented by $30, or a nine by $39). The same comments apply to system routine FPRINTF_.

HSTOB_ requires a buffer to hold the original hex string, and a buffer for the new counting number. You may use one buffer for both, stuffing the new counting number produced by HSTOB_ on top of the old in the same location.

BTOHS_ ends the converted hex string with a null; unlike HSTOB_, it won't allow you to use the same buffer for the counting number you convert and for the new hex string which BTOHS_ produces.

ITOS_ ends a converted decimal string with a null. It fails to convert counting numbers larger than 32767 to a decimal string.

ITOHS_ converts any counting number up to 65535 to its hex string equivalent.

System routine HEX_ converts any single character of a hex string into a counting number. The character "D", for example, appears in the monitor as $44; after conversion, it appears in the monitor as itself: d.

Be warned that all SuperPET conversion routines which output decimal integers will fail on any values larger than 32767; after this value is reached, you may get a negative sign as output, but no number, or you may get a negative value. If you receive a negative value, add it to 65536, and you should have the right positive counting number.

We've annotated our Development manual to show the input expected by each system routine (pp. 172-175) and its output, and suggest readers do the same.

We print below two routines which convert binary notation because there are none we know of in the system library. If anybody can make 'em shorter, send 'em in!

```
; This routine converts a binary string of 8 digits into a counting number
; internally. For example, "01100110" is converted to the counting number $66.

bin2cn  ldx  #cur_dig    ; cur_dig is a buffer for the binary string.
        lda  #%10000000  ; Set initial form of mask byte.
        pshs a
        clrb
        loop
          lda  ,x+       ; Get binary string character.
          quif eq        ; Stop on end string null.
          suba #$30      ; Convert to a 1 or a 0.
          lsra           ; Put 0 or 1 into Carry Flag.
          if cs          ; If the Carry Flag is 1,
```

```
        orb ,s          ; set appropriate bit in B register.
      endif
        lsr ,s          ; Rotate the "1" rightward one bit in the mask byte.
      endloop
      leas 1,s
      clra
      std  cur_dig      ; Store converted counting number in buffer.
      rts

;Converts counting number to binary string. Will handle 8 or 16 bit numbers.
;Must be given a two-byte counting number, even if the high byte is clear.

cn2bin  clra                ; Shove null on stack to end converted string.
        pshs a
        ldy  #cur_dig+1     ; Get low byte first.
        lda  #$30           ; Load string format 0 and 1
again   ldb  #$31
        ldx  #0008          ; Bein' out of registers, we use X for decrement.
        loop
          lsr ,y            ; Put first bit in Carry Flag.
          if cs             ; If it's a 1,
            pshs b          ; Stack an ASCII 1
          else
            pshs a          ; Otherwise, stack ASCII 0
          endif
          leax -1,x         ; Decrement the count
        until eq
        tst cur_dig         ; Have anything in the high byte?
        if ne
          ldb cur_dig
          stb cur_dig+1     ; Move it to low byte.
          clr cur_dig       ; And clear so we don't loop forever.
          bra again
        endif
        ldx  #cur_dig
        loop                ; Pull decimal string off stack,
          puls b            ; store in cur_dig.
          stb  ,x+
        until eq            ; Null on stack ends string and loop, sets stack
        rts                 ; for RTS.
```

JUDGE CRATER AND FPPLIB.EXP For the first six months we owned SuperPET, we considered the case of the missing Judge to be no deeper a mystery than the addresses shown in the file "fpplib.exp" on our language disk. Then we got a copy of "Waterloo microSystems SuperPET Specifics" and found out that "fpp" means "floating point", though the second "p" in the acronym is still a mystery. The addresses in "fpplib.exp" are for the decimal math routines in SuperPET's ROMs. We summarize the function of each such routine below, and append data on some added routines discovered by John Toebes. We do not list addresses for those routines whose addresses are in fpplib.exp.

OK = 0
Underflow = 1
Overflow = 2

You'll find a status byte at $0087, named FPSTATUS_. It is never cleared by internal routines. To use it, clear it and examine it in your program. FPSTATUS_ reports the conditions

Divide by 0 = 3 shown at the left for all FP system routines.
Bad Argument = 4

FAC1 (at $80) and FAC2 ($90) are two Floating Point Accumulators (working registers) Waterloo set aside for computing floating point results. Most of the FP (floating point) routines use one or both. We now define what the routines do:

Decimal Arithmetic and General Routines

FLOAD_ loads FAC1 with the FP number pointed to by P1 (an address placed in the D register). The FP number must be in 5-byte FP representation.

FLOAD2_ loads FAC2 with the FP number pointed to by P1.

FSTORE_ stores the contents of FAC1, starting at the address given in P1.

FZERO_ sets FAC1 to zero.

TRF1F2_ transfers the contents of FAC1 to FAC2.

CNVIF_ converts an integer passed as P1 to FP and stores the result in FAC1.

CNVFI_ converts FP number in FAC1 to an integer returned in the D register.

EXPONENT_ returns the exponent of the number in FAC1 in base 2 without excess 128 notation; the result returns as an integer in D register.

FRACTION_ a number in FAC1 is converted (in FAC1) to the fractional part of the number. D register holds 0 if no fractional part exists, or the exponent of the number if a fraction remains in FAC1.

FADD_ adds the numbers in FAC1 and FAC2 and stores result in FAC1. The number in FAC2 is destroyed. FADD_ will add negative numbers (in effect, subtracting them from any positive total), as will FADD2_, following.

FADD2_ loads FAC2 with a number pointed to by P1, adds the numbers in FAC1 and FAC2, and stores the result in FAC1. The number in FAC2 is destroyed.

FSUB_ subtracts FAC2 from FAC1 and stores the result in FAC1, destroying the number in FAC2. Do not subtract negative numbers simply because they are negative (the effect of FSUB_ is to add them).

FSUB2_ loads FAC2 with the number pointed to by P1, subtracts FAC2 from FAC1, and stores the result in FAC1, destroying the number in FAC2.

FCMP_ compares FAC1 with FAC2 and returns the result in D Register: -1 ($FFFF) shows FAC1 < FAC2; 0 shows FAC1=FAC2, +1 shows FAC1 > FAC2. Numbers in FAC1 and FAC2 are not changed.

FTEST_ tests the number in FAC1 and returns results in D register: -1 ($FFFF) shows FAC1 < 0; 0 shows FAC1=0; +1 that FAC1 > 0.

FMUL_ multiplies FAC1 by FAC2, stores the result in FAC1; destroys the number number in FAC2.

FMULBY_ loads FAC2 with a number pointed to by P1, multiplies FAC1 by FAC2, and stores the result in FAC1. Destroys the number in FAC2.

FMUL10_ multiplies FAC1 by 10, stores result in FAC1, destroys FAC2 number.

FDIV_ divides FAC1 by FAC2, stores result in FAC1, destroys FAC2 number.

FDIVBY_ loads FAC2 with a number pointed to by P1, divides FAC1 by FAC2, stores result in FAC1, destroys number in FAC2.

FDIV10_ divides FAC1 by 10, stores result in FAC1, destroys number in FAC2.

FNEG_ negates the number in FAC1 (converts from positive to negative form, and vice-versa).

FADDHALF_ adds 0.5 to FAC1 and destroys the number in FAC2.

FFLOOR_ calculates the FP representation of the largest integer not greater than the number in FAC1, stores result in FAC1; FAC2 not affected.

CNVF2S_ converts number in FAC1 to string format, and places the string in a buffer whose address is stated as P1. Stores a null byte at endstring in the buffer. D register returns the address of this null byte.

CNVS2F_ converts a number in string format to FP; the address of the string is P1. Waterloo says P2 is the address where scanning of the number is to stop, but we are equally successful in stuffing in a dummy value (see left). John Toebes says the routine stops either at the endstring null or at a real address (P2), so the trick at left is valid. We didn't push 0000 because you can foul up CC register flags when the stack value is used in the routine. The result of CNVS2F_ is stored in FAC1. Warning: FAC2 condition unpredictable. The address of char following the last one converted returns in D register. Note: the address for STOP (P2) must point one byte beyond the entry.

```
ldd  #0005    ; P2 a dummy
pshs d        ;
ldd  #string  ;
jsr  cnvs2f_  ; P1 address
```

To demonstrate some of the routines above, we show below a simple way to sum (add positive numbers or subtract negatives) for a cumulative total. A user buffer named BUFFP holds the previously accumulated total:

```
    ldd  #0005      ; Load the dummy (the real stopping address may be used)
    pshs d          ; Stack it
    ldd  #new_num   ; P1 is starting address of number in string format
    jsr  cnvs2f_    ; Convert it to FP format, store in FAC1
    ldd  #buffp     ; P1 is address of buffer for cumulative total in FP format
    pshs d          ; We'll need it again.
    jsr  fadd2_     ; Load buffer total into FAC2; add it to FAC1.
    puls d          ; P1 address of cumulative buffer.
    jsr  fstore_    ; Move new total from FAC1 into cumulative buffer
    leas  2,s       ; Recover on stack

buffp  rmb 5        ; 40 bits (5 bytes) needed for external FP representation.
```

With his usual ingenuity, John Toebes has discovered additional FP routines in ROM, which he defines below. The first five routines are identical to similarly named routines above, except that P1 is placed in X register, not in D:

FLOADX_ $A2BE. Same as FLOAD_.
FLOAD2X_ $A2DD. Same as FLOAD2_.
FSTOREX_ $A2FC. Same as FSTORE_.
FADDX_ $A43C. Same as FADD_.
FMULBYX_ $A4C5 Same as FMULBY_.

QKTRF1F2_ $A389. Same as TRF1F2_, except the guard byte is not thrown away.

KILGUARD_ $A3CD. Removes any guard bits that may be set in FAC1. Essentially rounds FAC1 up and normalizes it, which FROUNDUP_ does not do.

FROUNDUP_ $A3D5. Rounds FAC1 to 4 bytes of precision (external format). The CC register is set to reflect any overflow.

CHKGUARD_ $A4A8. Remove any guard bits from FAC1 in preparation for saving it in external format.

COMPFAC1_ $A3BC. Complements the value in FAC1.

FMULBYTE_ $A4FB. Useful for fancy work. Multiplies FAC2 by a byte in B register; adds result at location pointed to by X register (normally, FAC1).

The following two routines perform similarly, adding/subtracting FAC1 and FAC2 without touching the exponents:

FADDAC_ $A613 FSUBAC_ $A644

ADDECAC1_ $A147. Add a 4 byte double-precision integer to the current contents of FAC1 without changing the exponent. Pass address of integer as P1 in D register.

FLOADMAX_. $A2B9. Loads MAXREAL into FAC1. MAXREAL is the largest FP value which can be handled in SuperPET.

CHKDIVAC_ $A5C5. Compares FAC1 and FAC2; sets carry bit if FAC1>FAC2, without affecting the exponents.

Transcendental Routines and Such

In the next three routines, the argument must be in FAC1 before the routine is called. Angles must be expressed in RADIANS (in FAC1). Results return in FAC1; FAC2 is destroyed. ATN_ returns results in radians in FAC1.

COS_ Returns cosine SIN_ Returns sine ATN_ Returns arctangent

The remaining routines return the results shown below:

SQR_ returns in FAC1 the square root of a number placed in FAC1. FPSTATUS is set to BAD ARG if the argument is negative.

LOG_ returns in FAC1 the natural log (base e) of the argument in FAC1. FPSTATUS is set to BAD ARG if the argument is equal to or less than zero.

EXP_ returns in FAC1 the value of e raised to the power of the number in FAC1.

POWER_ returns in FAC1 the value of X (in FAC1) raised to the power Y (in FAC2), FP STATUS is set to BAD ARG if X is negative or Y is not an integer.

FLOATING POINT ROUTINES IN SUPERPET
by John Toebes, VIII

When we look at how floating point was implemented on the SuperPET, we can only wonder why it was done the way it was. The routines are very similar to the 6502 floating point routines written for the Apple II. The similarity shows up in the absence of code to use SuperPET's Y register as well as in no code using the A:B register pair as a true 16-bit register. This, coupled with the absence of many of the powerful addressing modes of the 6809, boils down to one simple fact: the floating point routines in SPET are both slow and inefficient.

As does the Apple II, the SuperPET uses two Floating Point Accumulators, called FAC1 and FAC2, at $0080 and $0090, respectively. Floating point numbers that have been loaded into an accumulator have a <u>different</u> format from those not in an accumulator. The majority of the routines require that you first load FAC1 or FAC2. However, due to the design of the code, it is possible to store a num-in <u>internal</u> (accumulator) format outside an accumulator and then to transfer or process it later, passing the address of your number in the X register.

Numbers in <u>external</u> format (for loading into an accumulator) take up five bytes of memory. The first bit of the first byte is the sign bit for the number. The next eight bits (7 from the first byte and the first one from the second byte) specify the exponent in excess 128 notation; a value of zero indicates that the value of the floating point number is zero. The remaining 31 bits form the mantissa, which is treated as a 32-bit binary fraction, where the first bit always is one. Note that this is contrary to the documentation in Appendix A (Page 65) of the System Overview Manual.

To illustrate, looking at the number in memory as a series of bits, we have the following typical representation, where S is the sign, E the Exponent, and M the Mantissa:

```
SEEE EEEE   EMMM MMMM   MMMM MMMM   MMMM MMMM   MMMM MMMM
  Byte 1      Byte 2      Byte 3      Byte 4      Byte 5
```

A value of 1 in the sign bit indicates a negative number, while a value of 0 indicates a positive number.

EEEEEEE is the exponent in excess 128 notation, obtained by adding 128 to the exponent; an exponent of 0 is thus converted to 10000000 in binary. The sign bit strips the 1; the exponent is 0. The exponent is expressed in powers of 2, not 10, and indicates the power of 2 by which the mantissa must be multiplied to express the value of the number. The largest exponent thus possible is 2**127 or 1.7E+38 (represented as $FF or 128+127) while the smallest exponent is 2**(-127) or 5.9E-39 (represented as $01 or 128-127).

MMMMMMM MMMMMMMM MMMMMMMM MMMMMMMM is the mantissa, represented as a binary fraction with a 1 as the first bit. Because the first bit is always 1 in this binary fraction, the largest possible mantissa is .FFFFFFFF (in hex), or .9999999999999995343387 (1-2**(-31)) and the smallest is .8000000 (hex) or 1/2.

Since the first bit is assumed to be a 1, 1/2 would be stored as all zeros in the mantissa. With the requirement that the mantissa be a fraction with the first bit always 1, we have what is commonly referred to as a normalized number. Since normalization is done after any mathematical operation, any number that has been stored in memory can be assumed to be normalized; we thus do not need to store that first bit.

Combining the three parts to produce a number is a bit tricky because everything is represented as a power of 2. Mathematically, the number can be expressed as

```
-1 * SIGN * MANTISSA * (2 ** (EXPONENT - 128))
```

taking into account the representation of the mantissa. When you want to take apart a number, you must first go through the tedious task of converting the mantissa to its equivalent decimal fraction, and then multiplying it by 2 raised to the exponent. With a good calculator, you can do it quickly, but doing it by hand except in the simplest cases is a time-consuming task. Follow are a few examples of numbers stored in this format:

```
1   = .5          * 2**1  = 40 80 00 00 00
.5  = .5          * 2**0  = 40 00 00 00 00
.25 = .5          * 2**-1 = 3F 80 00 00 00
PI  = .785398163 * 2**-2 = 41 49 OF DA A2
```

To convert any standard floating point number to this notation, first set the sign bit to 1 if the number is negative. Then take the absolute value of the number and go through the process of normalizing it to a power of 2. The easiest way to do this is to continually divide (or multiply) the number by 2 until the resulting value is representable by the mantissa (between .5 and 1), counting the number of divisions that it takes. Use the count of the number of divisions or multiplications to form the exponent. If you divided, add 128 to the total number of divisions; otherwise, subtract the number of multiplications from 128 and use this for the exponent. Convert the result from your divisions/multiplications to a binary fraction, throw away the first bit (which must be a 1 if all went well) and use the leftmost 31 bits (padding with 0's on the right) as the mantissa.

For example, we convert 123.25 to its representation. First, note that it is positive, so the sign bit will be 0. It takes 7 divisions by 2 to get down to .962890625, so the mantissa will be 128+7=135 or $87. Converting the decimal .962890625 to binary produces: .111101101 binary; when you drop the first bit and pad to make 31 bits, you are left with the mantissa found on the next line: 1110110100000000000000000000000. When we put sign, exponent, and mantissa together, we get the following binary entry:

```
     S EEE EEEE Mantissa--------------------------------
     0 100 0010 1 111 0110 1000 0000 0000 0000 0000 0000  (hex 42 F6 80 00 00)
Hex:   4   2      F    6    8    0    0    0    0    0
```

Internal numbers (usually in an accumulator) take up 7 bytes of memory. The entire first byte indicates the sign of the number. The next byte specifies the exponent in the same excess 128 notation as the external format, with a value of zero indicating that the value of the entire floating point number is zero. The remaining 5 bytes form the mantissa, treated as a 40-bit binary fraction.

If you look at a normalized number loaded in an accumulator, you will see:

0000000S EEEEEEEE MMMMMMMM MMMMMMMM MMMMMMMM MMMMMMMM MMMMMMMM

where S is the sign bit. A 1 indicates a negative number.

EEEEEEEE is the exponent in the same representation as an external format number. Note that 0 indicates that the entire number is zero.

MMM...M is the mantissa of the number represented as a NORMALIZED binary fraction (the first bit is always 1). The last byte is the 'guard' byte to allow for greater precision in performing calculations. It is used if the number should be rounded when an internal format number is converted to an external format number.

Converting an internal format number to and from Decimal is the same as for an external format number except that there are more bits in the mantissa, and, most importantly, the first bit of the mantissa is stored as a 1 instead of being assumed. In this case, 1/2 is stored as $80 00 00 00 00 in the mantissa, instead of the zeros that the external format uses.

CC REGISTER FLAGS AFTER MATH
Reference Data

When the 6809 performs arithmetic, either directly or with Waterloo routines, you have only one way to learn if the results are right or wrong--and that's to check the condition of four CC (Condition Code Register) flags. We summarize the condition of each pertinent flag below for all of the integer arithmetic operations for a reasonable set of values:

CC REGISTER CONDITIONS AFTER ADDITION

If total, after addition, is that below, then flags will be as shown:	SIGN	ZERO	OVERFLOW	CARRY
1. At or below 32767 ($7fff)	0	0	0	0
2. Above 32767 (from $8000 to $FFFF)	1	0	1	0
3. At 65536 ($10000)	0	1	0	1
4. Above 65536 (above $10000)	0	0	0	1
(Patterns well above 65536 not checked)				

CC REGISTER CONDITIONS AFTER SUBTRACTION

If total, after subtraction, is that below, then flags will be as shown:	SIGN	ZERO	OVERFLOW	CARRY
From 65535 to 32768 ($FFFF to $8000)	1	0	0	0
From 32767 to 1 ($7fff to 1)	0	0	0	0
At zero	0	1	0	0
From -1 to -32767 (to -$8001)	1	0	0	1
From -32768 to -65534 (-$8000 to -$FFFE)	1	0	1	1
At -65535 (-$FFFF)	0	0	0	1

STATUS OF CC REGISTERS IN RETURN FROM SYSTEM ROUTINE ___MUL

If total, after multiplication, is that

below, then flags will be as shown:	SIGN	ZERO	OVERFLOW	CARRY
From 1 to 32767	0	0	0	0
From 32768 to 65534	1	0	0	1
At 65535	1	0	0	0
At 65536	0	1	0	0
From 65538 to 98302	0	0	0	1
At 98304. Cycle begins to repeat.	1	0	0	1

STATUS OF CC REGISTERS IN RETURN FROM SYSTEM ROUTINE ___DIV

In division, for values or results shown, the flags will be as shown:	SIGN	ZERO	OVERFLOW	CARRY
Division: $FFFF/1. Result=FFFF	1	0	0	0
Division: $FFFF/2. Result=0000	0	1	0	0
Division: $FFFE/2. Result=FFFF	1	0	0	0
From 32767 to 1 (e.g., $7FFF/$80)	0	0	0	0
At zero ($0002/$7FFF or similar nonsense)	0	1	0	0

HOW ONE FAMILY USES SUPERPET
Background and Outlook

by Delton B. Richardson
4299 Old Bridge Lane
Norcross, GA 30092

Even though I have worked for some years developing software for telephone systems, I hadn't really used or known much about microcomputers until three years ago. I had often wished for the power of a computer at home, although the only actual needs I could identify were for a simple word processor and a way to account for my personal finances.

A little over three years ago I agreed to help a good friend look for a computer system for a small wholesale business he'd started. He wanted one that didn't cost too much, would have power enough for his present needs, and would allow for future growth. At that time, the 8050 disk drive with one megabyte, and the 8032 with a built-in 80-column monitor provided the two most important features, and the price was right.

We found a general-purpose software package for inventory, invoices, accounts receivable and payable, and general ledger. It had many deficiencies, but was very unusual, for it was written in BASIC 4.0 and so could be enhanced or modified quite easily. I decided to help my friend and learn about microcomputers by enhancing the software myself.

Over the next year, his business grew quickly; we changed the software often; I rewrote the sales invoice program for larger and better invoices. We stored the invoice data on disk so we could postpone until the end of the business day the processing of that data to general ledger, the accounts, and to inventory files. Computer hardware prices meanwhile fell dramatically; my friend bought additional systems. He then ran three computers to accomodate his peak loads and to protect himself against hardware failures. We also enhanced much of the software; we expanded files for more accounts and increased the inventory list to 6000 items. Last, we modified the programs to use the 8250 disk drive.

After all this, I had a pretty good idea about the capabilities of micromputers on the market at that time. The second system we bought was a SuperPET, since the price difference was small and it was fully compatible with the 8032 soft-

ware we already used: WordPro 4+ for word-processing and The Manager as a simple but rather powerful data base manager. SuperPET gave us an improved BASIC; programs were easier to develop. We also wanted to use APL for quick and simple programs, though at first we had no way to print APL to our 8023 printers [Ed. If you came in late, Delton wrote the programs on the ISPUG APL-to-printer disk for the Commodore 8023 printer.] Our only major disappointment was the slowness of the Waterloo languages compared with BASIC 4.0.

About this time, I decided that SuperPET had all the power I needed for use at home for the foreseeable future and purchased one. I now did word-processing at home; my wife was doing accounting for our friend's business and she could now bring disks and do much of the work at home. I then completed a microBASIC project to handle home accounting. We were then able to use the computer for all its intended purposes. Except for a bit of trouble while in warranty, SuperPET has worked well.

While I was developing programs, the computer was in heavy use. Lately, I have used it just to run applications; one or twice a week to process words; once a month to update our home accounting books (a job it does neatly, replacing a tedious, fragmented, error-prone, time-consuming and hated task). We have some other uses; a decision-making program in APL, and Manager database files for my investments and stocks.

For a while I used Manager files at home to keep track of development work at the office and to print monthly reports; it was far easier to use SuperPET than the mainframe at work. Several months ago, the office got an IBM PC, so I'll move this work to that computer, where it'll be more convenient.

The office is now adding a dial-up facility for our mainframe, so my latest use for SuperPET is as a terminal at home. I'm going to use COM-MASTER at 1200 bps to get and send the files. I know that in time newer, cheaper, and more powerful computers and new, attractive software will become available. SuperPET is slower than newer machines and without color; it'll never handle the large, new programs now on the market. Yet, to me, SuperPET will not be obsolete until it will not perform the tasks I want done or it fails beyond repair. For now, I am happy with it; it meets all my computing needs, and I have no intention of replacing it. I'm satisfied with the software now at hand. Sometimes, though, we are surprised by new ideas. I hope ISPUG remains healthy and that members will continue to develop new approaches and new applications.

FILE TRANSFER USING A NULL MODEM
by John Frost
Associate Editor

Have you ever wanted to transfer SuperPET files to or from another computer, but been frustrated by incompatible disk formats? How about trying a direct cable connection through a null modem? This simple device connects the SuperPET's RS-232 port to a similar port on the other and lets you transfer ASCII files between the two as if they were connected in a normal telephone/modem hookup. The null modem file transfer requires each computer to have modem software. I found the SPET telecom package NEWTERM ideal for the Commodore side and used the popular PC-TALK III (public domain freeware) for my COMPAQ portable. Note: If you do not have the NEWTERM software, you may transfer a file in SuperPET's microEditor with the PUT SERIAL command. [Ed. NEWTERM and other SuperPET telecommunication programs with instructions are available from ISPUG. See address, last page; cost is $15 U.S. for 6809 capability, either in 4040 or 8050 format.]

A null modem is basically an interconnecting cable that directs the output of SPET to the input of the other computer, and the output of the other computer to the SuperPET input. In addition, all necessary hardware handshake functions for the two computers, normally provided by the modems, are satisfied by a series of wiring jumpers within the cable.

The null modem can take a variety of forms and usually reflects what materials are at hand. I use back-to-back solder type connectors held together with spacer hardware (stand-offs) making the wiring interconnects very short. The completed device is only about 2 1/2 inches long. The null modem is connected to each of the machines with standard ribbon cable and connectors. You may use either pin or socket connectors according to your particular requirements.

A diagram of null modem wiring that has worked well for me is shown left, below.

```
SPET                          Other Computer

Pin 1 -------- Chassis Ground----- Pin 1
    2 -------- Data from SPET -------- 3
    3 -------- Data  to  SPET -------- 2
    4 --> A                      C <-- 6
    5 --> A                      C <-- 8
    6 --> B                      C <- 20
    8 --> B
   20 -> B
    7 --------  SIGNAL  GND ---------- 7
```

In my case, the other computer is a COMPAQ with the RS-232 port provided by an AST "Mega-Plus" plug-in board. The jumpering at the "other" computer connector is representative of that required by most RS-232 ports, but check your manuals for your specific application.

Jumper the pins marked A together; do the same for B and C pins. Keep all A's separate from B's, etc. The wiring jumpers allow SPET and the other computer to provide their own RS-232 enabling signals. SPET's "Request to Send" (Pin 4) provides the required "Clear to Send" (Pin 5); similarly the "Data Terminal-Ready" (Pin 20) provides both the "Data Set Ready" (Pin 6) and the "Carrier-Detect" (Pin 8). My early, three-board SPET is very sensitive to these jumpers and I use them for all my RS-232 connections. You might get away with fewer jumpers on later models, however I recommend the full complement as a start. [Ed. According to Waterloo manuals, you need the jumpers on any model.]

With the null modem in place, properly connected to the SPET and to your communication partner, we can begin a file transfer. First check the baud rate with the SETUP function; make sure both machines are configured for the same speed.

Load and execute the modem software on both machines and try some simple keyboard exchanges. The SPET Editor TALK command is sufficient for these keyboard exchanges if the NEWTERM package is not available. It is possible that the machines will not display their own keyboard entries; you may have to toggle the echo or echo-plex command. NEWTERM toggles the local echo with the PF7 key. Make sure that both machines are configured for the same parity and word length; any differences usually show up as garbled messages.

A file transfer from SPET (with NEWTERM) is begun with the PF9 key; you receive a file with PF8. With appropriate software and coordinated keystrokes on the "other machine", file transfers should be a breeze.

This article was prepared on a COMPAQ using a popular word processing package. The file was converted to ASCII and transferred to SPET through the null modem. A diskette containing the file, now in Commodore 8050 format, was submitted to the Gazette.

If you wish to transfer binary (PRG) files to and from the SPET, you must have telecom software that supports an "eight bit no parity protocol." COM-MASTER or PETCOM, previously reviewed in the Gazette, are suitable. I haven't attempted such a transfer and leave it as an exercise for the reader.

THE APL EXPRESS by REG BECK

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

When SuperPET arrived on the scene, Waterloo's was one of few APLs available; APL users of all varieties quickly bought SuperPETs. Now that faster and more powerful APLs are available, the professionals are moving on to other machines. The original purpose of SuperPET APL as a teaching and learning tool, somewhat obscured by the initial flurry of activity, has re-emerged. This is not meant to detract from the many and excellent applications we have seen during the past two or three years. The machine will continue to run these programs and more! THE APL EXPRESS strives to present a varied offering to readers, but the main thrust of the column is to help beginners, students and teachers to use the SuperPET APL operating system and to learn to program in APL.

Programs for accurately calculating sunrise and sunset times are available from various sources. These are of interest to navigators, astronomers and amateur radio operators, among others. Although complex programs are beyond the scope of this column, much may be learned from the development of simplified versions. Here are a fairly simple pair of equations which permit calculation of sunrise and sunset times anywhere in the world to an accuracy of plus or minus a few minutes:

$$\text{sunrise} = \frac{\text{long W}}{15} + \frac{\arccos(\tan \text{in} \cdot \tan \text{lat N})}{15}$$

in decimal Coordinated Universal Time (or Greenwich Mean Time)

$$\text{sunset} = \frac{\text{long W}}{15} - \frac{\arccos(\tan \text{in} \cdot \tan \text{lat N})}{15}$$

where: long W is the longitude of the location in decimal degrees West.
lat N is the latitude of the location in decimal degrees North.
in is the inclination of the Earth's axis with respect to the Earth-sun axis in decimal degrees North.

If the coordinates are East longitude, South latitude or South inclination, enter the angles with negative signs.

The inclinations may be obtained from The American Ephemeris and Nautical Almanac and vary for each day from a maximum of 23 degrees to a minimum of zero. For best accuracy the values should be those for the current year. Unfortunately, the table I have dates from 1977, which introduces some error. The following APL functions are useful in calculating the times:

```
DTR: ○ω÷180                    ⍝DEGREES TO RADIANS
RTD: ω×180÷○1                  ⍝RADIANS TO DEGREES
ROUND: .01×⌊.5+ω×100
TANPART: (3○ DTR α)×3○ DTR ω
SRT: ROUND (ω[1]+RTD ¯2○T)÷15:(T>1)∨(T←ω[2] TANPART ω[3])<¯1:'Δ'
```

```
SST: ROUND (ω[1]-RTD ¯2○T)÷15:(T>1)∨(T←ω[2] TANPART ω[3])<¯1:'Δ'
UT:R+((R<0)×2400)-((R←((⌊ω)×100)+⌊.5+(ω-⌊ω)×60)>2400)×2400:ω='Δ':'UNDEFINED'
SUNRISE: UT SRT ω
SUNSET: UT SST ω

⍝SST AND SRT ARE CONDITIONAL BECAUSE FOR LATITUDES GREATER THAN
⍝67 DEGREES THE SUN MAY NOT RISE OR SET ON A PARTICULAR DAY.  IN
⍝THAT CASE TANPART MAY BE GREATER THAN 1 OR LESS THAN -1 FOR
⍝WHICH VALUES ARCCOS IS NOT DEFINED.
⍝UT CALCULATES THE 2400 HOUR COORDINATED UNIVERSAL TIME.  SINCE
⍝THE RESULT MAY BE NEGATIVE OR GREATER THAN 2400 HOURS, 2400 IS
⍝ADDED OR SUBTRACTED DEPENDING ON THE VALUE OF R.
⍝TO RUN THE PROGRAM ENTER SUNRISE FOLLOWED BY THE WEST LONGITUDE,
⍝NORTH INCLINATION AND THE NORTH LATITUDE.  SAME FOR SUNSET.
⍝EXAMPLE--    SUNRISE 122 18 52.3  FOR MY LOCATION ON AUG 1ST.
```

The values of the inclination for late July, Aug. (1977, unfortunately) are:

July 18	21 N	Aug 1	18 N	Aug 12	15 N	Aug 22	12 N	Aug 30	9 N
July 24	20 N	Aug 5	17 N	Aug 15	14 N	Aug 25	11 N		
July 28	19 N	Aug 9	16 N	Aug 19	13 N	Aug 27	10 N		

If you have a current set, use them in the calculation. I will leave it as an exercise to convert the functions in direct definition to the del form. Try the program and compare with your sunset and sunrise times (available from the nearest airport).

* * *

APL separates assignment and equality. This is obvious to an APL programmer, but the beginner in APL, especially one who knows BASIC, may not see the reason for this at once. As I teach both BASIC and APL but only use APL these days I have to consciously separate the two and the different methods used in each when teaching. Some time ago, when discussing looping and subscripted variables in BASIC, I wrote the program segment at left on the blackboard...

```
100 READ A(I=I+1)
110 IF I=10 THEN 130
120 GOTO 100
```

When the students tried this (with appropriate DATA statements) they were unable to print out the list, A(I). After staring at the screen for a minute, I saw the error and made the appropriate correction. In BASIC, A(I=I+1) is the variable A(0) only, since I=I+1 asks "is I=I+1 ?" in this context. The answer always is a resounding NO. There is quite a difference in BASIC between the two statements I=I+1 and A(I=I+1). In APL we have:

```
I=I+1       EQUALITY WHICH RESULTS IN AN ANSWER OF NO (A ZERO IS RETURNED)
A[I←I+1]    ASSIGNMENT WHICH INCREMENTS THE COUNTER, I
```

You can see that the ambiguity has been eliminated in APL by separating these two fundamentally different operations.

Now that we are on the topic of differences between APL and other languages, let us look at how APL treats data. Typically, APL attempts to handle data in natural-sized pieces. These pieces are scalars, vectors (one dimensional arrays) of numbers or characters, and matrices (multi-dimensional arrays) of characters or numbers. These are also used in BASIC and Pascal. The difference is in the way the data is manipulated. In APL, the whole vector or array is typically manipu-

lated, while in the linear languages you must manipulate the individual elements of the array as scalars (the linear languages are also called scalar languages). This accounts for part of the 10 to one reduction in code achieved, on the average, when APL is used vs. a linear language. It also accounts for much of the appeal of APL to engineers and other technical people who want to get something working quickly. It also appeals to those with matrix math backgrounds. Keep BASIC and standard Pascal in mind as you work through the following examples, which demonstrates APL's efficiency in dealing with vector data. [Ed. We could not print Reg's example from his APL file without throwing our printer into an infinite loop, in which it repeated, ad nauseum, the last three lines. We modified the file to no avail. So we copied the file to "ieee4" until the printer began its infinite loop, shut it off, trimmed the copy, and pasted it in the final copy. APL somehow managed to send either an ESC or CONTROL sequence our printer could not handle. The language is never easy to print from a text formatting program; sometimes it is impossible. Innovation has its perils.]

```
V←⎕                              ⍝ENTRY OF THE VECTOR
⎕:
      2 5 8 7 ¯4 ¯5 1 6          ⍝UP TO ONE SCREEN LINE
      V                          ⍝PRINTS OUT ALL THE DATA AT ONCE
2 5 8 7 ¯4 ¯5 1 6
      ⌽V                         ⍝REVERSES THE ORDER OF THE DATA
6 1 ¯5 ¯4 7 8 5 2
      V[⍒V]                      ⍝SORTS THE DATA IN DESCENDING ORDER
8 7 6 5 2 1 ¯4 ¯5
      V[⍋V]                      ⍝SORTS IN ASCENDING ORDER
¯5 ¯4 1 2 5 6 7 8
      ⍴V                         ⍝COUNTS THE NUMBER OF ELEMENTS IN V
8
      ⍴(V<0)/V                   ⍝COUNTS THE NUMBER OF NEGATIVE ELEMENTS
2
      +/V                        ⍝FINDS THE SUM OF THE DATA IN V
20
      1∊V                        ⍝IS 1 AN ELEMENT OF V?
1
      3∊V                        ⍝IS 3 AN ELEMENT OF V?
0
```

A favourite slogan of certain professional APL programmers is "Linear languages cause brain damage."

When teaching APL an effective mechanism for promoting "APL style" is to force the use of one-liners. A "one-liner" is a non-recursive function with no occurences of right arrow. Direct function definition is ideal for this purpose. Most students taking APL probably have learned a scalar language and tend to "write BASIC" in APL. This can be overcome by forcing one-liners. This and other ideas on teaching APL are discussed in a paper, "The Use of APL in Teaching Computer Science," by Lawrence Snyder. It may still be available from the Department of Computer Science, Yale University, New Haven, Conn. Although the first language I learned was APL, I was forced to abandon it for several years for want of a machine. An old 8K PET, bought in 1978, offered BASIC. Upon my return to APL with the purchase of SuperPET, the problem of "writing BASIC" in APL restricted my progress. I still suffer from it from time to time, as you may note in some of the examples in this column. One-liners help; the following example is from

Gilman and Rose, "APL: An Interactive Approach," 2nd Edition revised, 1976; it is followed by the result of applying the program, TABULATE, to a sentence:

```
TABULATE:(ρA)ρ(,A←A∘.≥ι⌈/0,A←(A≠0)/A←A-1+0,¯1↓A←A/ιρA)\(~A←T∊' .,;:')/T←ω,' '

TABULATE 'FIRST LESSON: NAMES, EXPRESSIONS AND SOME PRIMITIVE FUNCTIONS.'
FIRST
LESSON
NAMES
EXPRESSIONS
AND
SOME
PRIMITIVE
FUNCTIONS
```

It's instructive to dissect TABULATE to see how it works. After you have done this, think about writing it in BASIC with a loop and an IF-THEN statement to check for ASC values. It's not difficult in BASIC and the great temptation is to duplicate that in APL; i.e, to "write BASIC". TABULATE creates a logical vector of the same length as the sentence. It has ones where the spaces and the punctuation are and zeros elsewhere. The complement of this vector is used to compress spaces and punctuation out of the sentence. The vector is then used to find the lengths of the words in the sentence. A logical array is then generated and used to insert spaces between the words so that each word with spaces is equal in length to the longest word. The expanded sentence is then formed into an array using dyadic rho.

TABULATE is an example of an APL "idiom," the term for a construction in a programming language of a logical operation for which the language possesses no primitive. TABULATE is a tabular structure which uses an array as a vector; the rows are the elements. Some other idioms are:

```
(U≠V)/V          ⍝REMOVE OCCURANCES OF U IN VECTOR V
((VιV)=ιρV)/V    ⍝REMOVE DUPLICATES FROM V

⍝A QUOTE FROM SNYDER: THE ACQUISITION OF THE 'APL STYLE' IS PRIMARILY
⍝A MATTER OF LEARNING IDIOMS FOR COMMON OPERATIONS.
```

A teacher can improve learning of APL by using interactive methods, a large video display and through the teaching of idioms. The "Finn APL Idiom Library" is available from APL Press, Suite 201, 220 California Ave., Palo Alto, California 94306.

SOMEWHERE IN THE TWILIGHT ZONE DEPT. For the past few years we've read a lot on the virtues of transportable code. You write this great program in 'C' or Pascal or some other high-level language and sell a jillion copies to folks who use an XXXXX microprocessor. Then a YYYYY micro comes on the market. Do you rewrite your program? Nah, you port it over to the YYYYY with a new compiler or code generator which converts the program to machine language on YYYYY. You never rewrite the program itself; all you need to generate object code is a new compiler/code generator and a bit of hard-writ assembly stuff to handle I/O and such. At least that's the theory. Somehow or other, it doesn't seem to work out quite that way in practice. First, you never sell a jillion copies--because the original code is too slow to compete against

hand-written assembly which does the same thing; second, the transported code on YYYYY is often slower than it was on XXXXX.

We note some examples: the operating system for Lisa I (now dead) was written in Pascal; BYTE and InfoWorld told us that just about any benchmark on Lisa I was slower than an Apple II. The original BASIC for Macintosh was, we're told, written in 'C'--and it ran more slowly than Applesoft. A spreadsheet named MBA was written in Pascal; it was similar to Lotus 1-2-3 (written in assembly); anybody heard of MBA recently? We know a couple of gurus who write everything in 'C' and claim that the code generated by their 'C' compiler is faster than the best code which could be writ by hand. When we snorted that no dumb machine and program could beat a good programmer, they demonstrated with an optimizing compiler on some short routines, which did write some pretty good code. A Fortran programmer showed us the same thing on an optimizing Fortran compiler with some of his demonstration routines. But--note the words "short" and "demonstration."

What happens when routines are complex or when the high-level language cannot cope conceptually with a problem? (write an operating system in Fortran, hmmm?), when the compiler/code generator isn't as optimum at writing or comparing code as it might be, or it doesn't take full advantage of the instruction set on the microprocessor for which it generates code? In short, what happens when the language is not suitable or the code generator isn't smart?

```
somewher    ldd 4,s
            pshs d
            ldd  8,s
            addd codebuff
            call bank79772
            leas 2,s

bank79772   pshs  d
            leas  -2,s
            ldd   6,s
            pshs  d
            clra
            ldx   4,s
            ldb   ,x
            tfr   b,a
            clrb
            addb  1,x
            adca  #0
            addd  ,s
            puls  x
            std   6,s
            tfr   a,b
            clra
            ldx   2,s
            stb   ,x
            ldd   6,s
            stb   1,x
            leas  4,s
            rts
```

You may determine the answer yourself. Load Waterloo's V1.0 or V1.1 microEditor; scroll from top to bottom of a file; try some universal changes. Note that the mED was written in WSL, a high-level language. Then perform the same functions in either John Toebes' V1.3 mED (on the ISPUG Utility Disk) or in Joe Bostic's BEDIT (ISPUG Utility Disk II), both of which were hand-written in assembly language. You behold the tortoise and the hare.

For a more specific example, see the code at left, disassembled from SPET's linker by John Toebes. As John says, "For those that doze off in the middle, let me replay this just a tad bit faster. Here is the inline code which does the same work; it needs no subroutine; 6 lines of code replace 28:"

```
somewher  ldd   8,s
          addd  codebuff
          tfr   d,x
          ldd   ,x
          addd  4,s
          std   ,x
```

Well, we think the point is made. Any time you hear that a software house has decided to write its code in 'C' or Pascal or some other HLL so as to reduce the cost of writing code and to make the code transportable, be cautious enough to give the programs from that house a trial for speed; benchmark before you buy. Hal Hardenbergh of Digital Acoustics recently noted that a new kilobuck AT&T

machine was slower at floating point calculations than a VIC 20. The FP routines most probably were written in AT&T's own language, 'C'...

REPAIR THAT COMPUTER
A Book Review
by Gary L. Ratliff

An increasing number of computer systems are being sold to both homes and businesses. The fact that you receive this newsletter reveals that you have invested a large amount of money in your system. With each computer come enough manuals to make you think you'll have to pursue a career in computer programming, but nary a word do they say about maintenance. Then something goes bad in the computer. At this point, you learn that having the system serviced by a pro can cost an arm and a leg. Yet much of what the pros does is routine maintenance, which the owner could learn to do for himself--if he only knew how.

While there is no shortage of computer programming texts on the market, books which address simple computer repair are rare. Those you can find usually assume such a degree of technical expertise that few readers can comprehend them. This sad state of affairs has come to an end with the appearance of the book, "The Plain English Repair and Maintenance Guide for Home Computers," by Henry F. Bechhold, published by the Computer Book Division of Simon & Schuster, available in bookstores for a mere $14.95. The book is well-written; the author does not assume the reader has prior knowledge of electronics. Of particular interest to us: the author owns a Commodore 8032, a very close cousin of SuperPET. Many of the examples presented are for the 8032 portion of the very machine we own.

The book begins at the beginning, with guidelines to repair. The reader is next instructed on the tools he should have if he expects to repair electronic gear. The third chapter covers what type of parts are needed and where you get them. The next portion of the text takes the reader into the guts of his computer; it shows how the various parts are used to make a working system, and how to read schemantic diagrams. Then follow instructions on routine cleaning and maintenanice of computer, cassette system, disk drives and printers. You are also told how to use a logic probe to trace out problems.

The final portion of the book shows some useful additions and modifications to a computer; these include a null modem and a simple breakout-box for the RS232 port/connection, among other easily-constructed items. Those who don't possess a logic probe are told how to build one. The appendices contain much useful information; one is a trouble-shooting guide to repairs you may easily undertake and those you had better leave to a professional. A bonus is a free consultation card which you may send to the author for one problem on which you need help.

In summary, you'll find this book easy to read and a help in both maintaining and repairing your computer system.

GLOBAL and LOCAL VARIABLES in Waterloo Languages
by John Seitz
Champlain Regional College
Lennoxville, Quebec

A major problem encountered with Waterloo software is that all variables except those used as formal parameters are global. This makes difficult the writing of procedures and functions for a program library. Any variables used in a procedure or function are global to a program--i.e., may be referenced by any line of code, anywhere. Thus the user of a library procedure must first check any variables used in it to make certain they are not used elsewhere in the program.

Formal parameters passed to a procedure or function are, of course, kept local. I illustrate such a formal parameter at left, below, where we pass a formal parameter to a procedure; the procedure accepts whatever value is passed under the procudure's local name of "accept_parm". The value of "accept_parm" will, however, remain local only if the variable name is not used in the main program. Its value is globally accessible--and not confined to either a procedure or function. In addition, the procedure or function itself may need internal variables which are not passed as parms. In ordinary usage, these are global, but in many cases, if not most, it would be desirable for them to be kept local.

```
...main program
call pass(parm)
...

proc pass(accept_parm)
 (accept_parm = parm)
```

I have found a simple technique which permits this. All variables used in a procedure or function must be declared as formal parameters in the header (i.e., in parentheses following the function or procedure name); the calling program must pass constants as "dummy" parameters to those which are local, while still passing actual parameters to the true formal parameters.

For example, assume a procedure named "pass" requires one formal parameter named "A" and that it uses internally another variable named "B". Assume we call this procedure normally with a formal parameter Z, as at the left. The problem is that if variable "B" is used elsewhere in the program, such use can affect operations in the procedure (and vice-versa). We may avoid this by the coding shown below. Here, the local variable "B" is set to zero by the dummy formal parm we have passed. The only requirement is that the dummy parms be of the same data type as those in the procedure or function header. Normally, we pass 0 as a dummy numeric parm or a null ("") as a dummy string parm.

```
...
call pass(Z)
...

proc pass(A)
...holds local var B
```

```
...
call pass(Z,0)
...
proc pass(A,B)
```

Steps for coding such procedures or functions follow: 1) Code the procedure or function; 2) list all variables, either as formal parms, local variables, or global variables; 3) create the header, specifying as formal parms first the true formal parms and then the local variables. Do not include any global variables; 4) instruct the user to call the procedure or function using formal parms for the true formal parms and constants as dummy parms for the local variables.

The demonstration below illustrates the effects you may obtain using the dummy-variable technique; while done in mBASIC, the technique applies equally to other languages. In summary, you may employ both truly local variables and global variables. Now, if someone can come up with a technique to pass a formal parm value back out of a procedure....

a = formal parameter variable with the same name in main program and procedure
b = formal parameter variable in main program, named "c" in procedure
c = formal parameter in procedure, named "b" in main program
d = global variable, common to both main program and procedure
e = local variable in both main program and procedure

A printout of the values of all variables is shown below:

```
 100 print ,," a  b  c  d  e"
 110 print "begin exec";
```

```
120 a=1 : b=1 : c=1 : d=1 : e=1
130 print ,, a; b; c; d; e
140 call pass(a,b,0,0)
150 print ,, a; b; c; d; e
160 print "add to all, main";
170 a=a+2 : b=b+2 : c=c+2 : d=d+2 : e=e+2
180 print , a; b; c; d; e
200 call pass(a,b,0,0)
210 print ,, a; b; c; d; e
220 print "end exec"
230 stop
240
250 proc pass(a,c,b,e)
260   print "enter proc";
270   print ,, a; b; c; d; e
280   print "proc: add to all";
290   a=a+2 : b=b+2 : c=c+2 : d=d+2 : e=e+2
300   print , a; b; c; d; e
310   print "leave proc";
320 endproc
```

	a	b	c	d	e
begin exec	1	1	1	1	1
enter proc	1	0	1	1	0
proc: add to all	3	2	3	3	2
leave proc	1	1	1	3	1
add to all, main	3	3	3	5	3
enter proc	3	0	3	5	0
proc: add to all	5	2	5	7	2
leave proc	3	3	3	7	3

[Ed. We have noted a few other approaches to this problem by programmers. One of the best and simplest places all local variables in capital letters, which are not used in a main program. Such local variables are made specific to one library function or procedure by numbering the library modules themselves and by using that number in the variable name, as in a library module named "proc printt", where the formal parms appear as shown at the left in library module 14. The combination of capitals and numbers indeed creates well-isolated local variables.]

```
proc printt(MAT SEND14$, LINES14)
```

RABBIT VS. TORTOISE
or
WAS DYSKTRA RIGHT?

A few years ago, someone (Dykstra?) said any that any programmer who learned BASIC was ruined for life. At the time, we thought he was bonkers. Now we are not so sure. In the past few years, we've received a lot of programs written in microBASIC which, despite the structure available, are full of GOSUBs and GOTOs. At first, we thought folks just hadn't had time to learn how to use structure; of late, however, we've decided that a lot of them don't even try. We've received some monumentally large microBASIC programs written as if structure didn't exist--and complaints from the writers thereof that microBasic is slow. If you persist in writing programs GOSUB and GOTO, it is slow. But--we conclusively prove below that microBASIC will run three times as fast if written to use structure and other built-in features. We demonstrate with a famous benchmark, the Sieve of Eratosthenes.

In the May, 1985 BYTE, Michael Vose benchmarked a new form of BASIC, True Basic, with BYTE's old GOTO benchmark, shown at left. He might as well have benchmarked a Porsche with a Model T ignition system. Why would anyone GOTO or GOSUB in a language which does not use line numbers unless you insist on them and in which structure is available? We dunno. Anyway, we ran the program in SPET in 505 seconds.

```
 1 ! BYTE's GOTO Benchmark.
 5 ! Executes in 505 seconds.
10 t = time : size = 7000
20 dim flags%(7001)
30 print "Start one iteration"
40 count = 0
50 for i = 0 to size
60   flags%(i) = 1
70 next i
80 for i = 0 to size
90   if flags%(i) = 0 then 170
```

Then we rewrote the benchmark, maintaining the spirit and purpose, and used only the simplest form of structure -- IF...ENDIF. We also used the MAT statement to set the 7001 flags to 1 at

```
100   prime = i + i + 3
110   k = i + prime
120   if k > size then 160
130   flags%(k) = 0
140   k = k + prime
150   goto 120
160   count=count + 1
170 next i
180 print "Time"; time-t;"seconds."; " Primes found"; count
```

the beginning of the program.

Using MAT cut execution time by 15 per cent; the insertion of one piece of structure cut it 37 per cent more. With a few simple changes, the structured code ran in 48 per cent of the time of the GOTO original--over twice as fast.

```
 90 !The sieve with MAT and structure
100 t = time : size = 7000
105 dim flags%(7001)
110 print "Start one iteration"
130 count% = 0
140 mat flags% = (1)
170 for i = 0 to size
180   if flags%(i)
190     prime = i + i + 3
195     k = i + prime
200     if k <= size
210       for j = k to size step prime
220         flags%(j) = 0
230       next j
240     endif
250     count = count + 1
260   endif
270 next i
290 print "Time"; time-t; "seconds."; " Primes found";count
```

Here is the structured version, without integers, for direct comparison. It runs in 243 seconds, less than half the time of the original, yet makes exactly as many calculations as the original. When we changed all the variables from real or decimal numbers to integers, run time dropped to 168 seconds, or to one-third the time required for the original. A reduction from 505 seconds to 168 seconds doesn't surprise us a bit. It demonstrates two things:

Structure speeds up execution!

GOTO/GOSUB programmers are locked into a terribly slow habit.

We summarize all tests in the table below, which demonstrates conclusively that GOTO programming is for turtles. All runs were in microBASIC V1.1.

Time to Run	BYTE GOTO Benchmark as written:	GOTO Benchmark with Integers	GOTO Benchmark with MAT and Integers	Structured Benchmark Using MAT With Reals	Structured Benchmark Using MAT With Integers
Seconds	505	391	343	243	168
Relative	1.0	0.77	0.68	0.48	0.33

The original benchmark runs in BASIC 4.0 in 292 seconds. If in 4.0 you take advantage of the inner FOR...NEXT loop which zeros the flags, the time drops to 193 seconds--which is slower than the structured mBASIC program. We sent a draft of this article to grand sachem Terry Peterson and bet him $25 he couldn't beat the microBASIC time in 4.0 if 1) he performed all calculations of the original, including the initial flag-set, and 2) he was limited to one statement per line in the working program (but not on the DIM and constant-definition line). He failed to beat the 168 seconds under the terms--but protested the terms were totally artificial, and sent back a BASIC 4.0 program optimized for speed which ran in 124 seconds, along with a HALGOL program, run with the MC 68000 microprocessor on the Digital Acoustics Grande board. HALGOL's time: 2.1 seconds!

Enough on (invidious) comparisons. We undertook to see if structure speeds up

programs; it does. Those who use microBASIC and want speed should employ structure. Don't believe the long-standing myth that structure is slow.

[See p 4, No. 1, Vol. II for more on HALGOL. Hal Hardenbergh, its creator, says it is the fastest interactive language in the world; we hope he and Terry send us a summary of how Hal's Grande board, power supply, and hookup to SuperPET are configured, the memory capacity options available, and the prices. We'd like to give readers a clear view of what is or soon will be available. Nick Solimene, one of our members, has both OS9 and the Grande plus HALGOL working in his SPET and reports no compatibility problems to date. HALGOL isn't finished yet.]

BITS BYTES & BUGS by Gary Ratliff, Sr.

215 Pemberton Drive, Pearl, Mississippi 39208

In our last treatment of the floating point routines in the Waterloo library we created a dyadic fuction which adds 2+3 to yield 5. It becomes the basis of our further exploration of SuperPET's math routines. When we stopped, last issue, FAC1 contained 3 and FAC2 contains 2. Clearly, by replacing the line reading JSR FADD_ with another dyadic function we'll be able to determine what the other routines do for us. So, let's begin.

If we replace the line JSR FADD_ with JSR FSUB_ then the answer will change from 5.000 to 1.000, for we'll have performed 3-2 = 1. There is no need to reassemble the program in the last issue to create a number of one-line changes; instead, we'll use the monitor. Enter it and ">l math.mod"; then ">t 1000-1030". You'll translate (disassemble) the program. When you see "JSR $a030", you have found the JSR to FADD_. Change the address of the JSR to that of FSUB_ by using the table printed last issue; we find FSUB_ at $a036. Simply modify memory with the "m" command: >m 1027 a036 (yes, the address is one byte beyond the JSR). Check for error by again "translating" 1026; you should see: JSR a036. If it is okay, "go" the code; a dump of the answer with: >d 1044 should show 1.000.

By similar change to the subroutines we call, we can explore in short order the functions performed by other library routines.

If we replace JSR $a030 with JSR $a003 we will call, not FADD_ but POWER_; the result of the calculation will not be 2+3 but 2^3, for an answer of 8.000. In a similar manner a change to FMUL_ will display the product, while FDIV_ will produce the result of 3/2. Try these and the other functions listed in the file fpplib.exp.

Unless you study material on the floating point routines printed elsewhere in this issue, the results may strange; in some of the functions the operations are performed on the numbers in FAC1 and FAC2, and in some the results return in the D register. We can use FCMP_ as an example; it compares the number in FAC1 to that in FAC2. The result of the comparison returns in D as -1 ($ffff) if FAC1 is < FAC2; 0 if the two are equal; or 0001 if FAC1 > FAC2. Modify the code in the monitor and try FCMP_. With 3 in FAC1 and 2 in FAC2, you should receive a 1.

This exercise can demonstrate the value of breakpoints (if you have not used them). If all we do is modify the JSR instruction at 1026 so that it performs a JSR CMP_ instead of JSR FADD_, the program continues to its end; we never do see the contents of the D register at the time FCMP_ does its work. We can stop the program by setting a breakpoint at $1029, with the command: >s 1029, at

which point the program will stop and give us a register dump.

Because the act of loading a parameter into FAC2 and calling a system routine is performed quite frequently, there is in the floating point library a series of routines which combine the loading of the parameter and the task of calling the right math routine. The result is a shorter program. In such routines, P1 (the parameter in the D register) points to the address of the argument we want.

Let us revise our math program to illustrate this technique. A program to take advantage of this shortcut is presented below: Compare this program with that presented last issue before you assemble and link.

```
XREF cnvs2f_, fstore_,  cnvf2s_, fadd2_

LDD  #end2   ; This is P2 for CNVS2F_, the end of the string.
PSHS d
LDD  #srt2   ; This is P1 for CNVS2F_, the starting address of the string.

JSR  cnvs2f_ ; Convert to floating point format in FAC1.
LEAS 2,s
LDD  #buf1   ; This is the location we use to store the converted number.
JSR  fstore_ ; We now have floating point form of 2 in buf1

LDD  #end3   ; This is P2, the end of the "00003" string.
PSHS d
LDD  #srt3   ; The starting address of 3
JSR cnvs2f_  ; Again, we convert to fp format in FAC1.
LEAS 2,s

LDD #buf1    ; P1 for FADD2_, the address of the number to be loaded in FAC2.
JSR fadd2_   ; Loads FAC2 with number 2, adds FAC1 and FAC2; result in FAC1.

LDD #buf2    ; Answer buffer.
JSR cnvf2s_  ; We convert the sum of 5 back to string format.
SWI

srt2 FCC "00002"
end2 EQU *
srt3 FCC "00003"
end3 EQU *
buf1 RMB 12
buf2 RMB 12
buf3 RMB 12
END
```

The .cmd file for this routine is the same as that in the last issue, except for the filenames:

```
"math2"
org $1000
include "disk/1.fpplib.exp"
"math2.b09"
```

Because this program is shorter than the previous one, our answers will appear in the answer buffer at $1041. You will find a number of combined function in the library: FADD2_: FSUB2_, FMULBY_, FDIVBY_; their addresses are listed in the last issue. All combine the operation of loading FAC2 and peforming a math operation; the results are identical to FADD_, FSUB_, FMUL_, and FDIV_.

Thus ends this second column on math routines of the Waterloo floating point package. The column is short because of the number of additional articles on the subject in this issue. I hope the material is useful and that those of you

who didn't know how to modify code in the monitor without reassembling are pleased with this time-saving trick.

THINGS I'D LIKE TO SEE IN THE NEXT VERSION OF mFORTRAN
or (with apologies to Lerner and Lowe)
Why Can't Our mFORTRAN be Like FORTRAN77?
by Stan Brockman, Associate Editor

Do you use your SPET in one powerful way for which it made--as a machine on which to write and debug code which you then upload to a bigger machine for production runs? Do you find that you can't easily do that with WATCOM's mFOR because of the extensive post-upload editing necessary before you can run it? I answer 'YES' to both these questions.

There are a number of differences between mFOR and FORTRAN77 as implemented on our VAX 780. What follows is correct to the best of my knowledge, but because my experience with the bigger computers is limited to the VAX, I may not be entirely consistent with FORTRAN77 as implemented on other machines.

My pet peeve is related to how character strings are implemented in mFOR. The mFOR character declaration statement simply defines a variable that may be assigned character data; FOR77 requires that the maximum length of a string also be defined. So what, you say? Well, the difference is not in itself a grave one, but the consequences of it are fairly far-reaching.

For starters, a SPET character variable is dynamic--it can be any length, up to available memory. Further, the length of the SPET variable can change each time a new string is assigned to it. The FOR77 variable can be only as long as its declared length--no more, no less. A character string shorter than the declared variable length will be be padded on the right with blanks before being stored; it will be truncated if it is too long. On output, the blank padding is retained (unless a substring of non-blank characters is extracted), which may not be what you intended.

This difference in storage mode carries through to the intrinsic function, LEN. mFOR will return the true length of a string assigned to a variable, while FOR77 will report the length of the variable as defined by the declaration statement. Aside from using a loop to count backward to the first non-blank character or using a VAX-dependent system function, there is no way to determine the true length of a string with FOR77--a real limitation, at times.

Well, let's look at the differences in getting character data assigned to the variables in the first place. mFOR permits list-directed input of character data (for instance, 'read *, ASTRING') which will then assign the characters between the beginning of the input line and the first delimiter (comma or end-line) to ASTRING. The same statement will crash a FOR77 program.

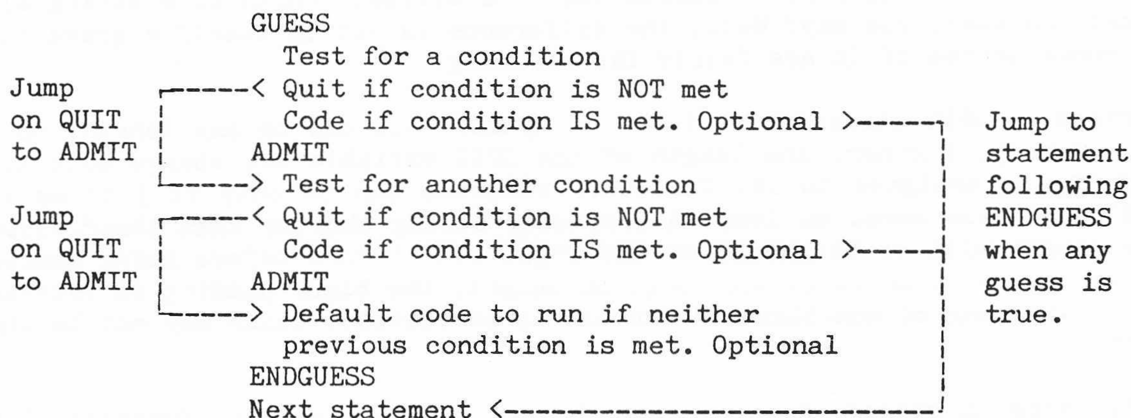
On the other hand, 'read 10, ASTRING', where the format statement is '10 format (a)', executed with FOR77, will read an entire input line (up to the maximum declared variable length), commas and all, and will assign the string to ASTRING. With mFOR, one of three things can occur: (1) if ASTRING has not previously been assigned a value, the first character only will be assigned to ASTRING, (2) if ASTRING has a previously-assigned value and the new string is at least as long as the old one, then that many characters from the new string will be assigned to the variable; the remaining characters will be truncated on the right, and, (3) if ASTRING has been previously assigned a value but the new string is less

than the previous length, a 'specified field width too big' error will occur.

As mentioned above, mFOR considers commas and end-of-line (or RETURN) to be delimiters for list-directed input. FOR77 also accepts blanks as delimiters for numeric input. In an earlier Gazette, it was observed that it would be nice if mFOR did not require commas between numeric input data fields. It surely would facilitate outputting data from the SPET to disk files if the commas didn't have to be explicitly included in order to delimit fields for subsequent list-directed reads.

Well, did I mention any of your pet peeves? Do you have others? Write them up, send them to our hard-working editor or to me. Better yet, write your own impassioned article. Maybe somebody out there will listen to us... [Ed. Stan's address is found on the very last page of this issue.]

WHY GUESS ABOUT GUESS...ENDGUESS? We continue to get questions on how to use the GUESS...ADMIT...ENDGUESS construct in SuperPET. It's found in most languages, including assembly, and is handiest when IF...ELSE...ENDIF won't cope well with multiple options. The basic concept:



We leave the construct whenever we do not QUIT a GUESS or an ADMIT. In short, whenever one of our guesses is right, we jump to the end of the structure. There is another way to employ GUESS...ENDGUESS, without an ADMIT, which is particularly useful in assembly language, though by no means limited to it. In the example below, we stuff commas into number strings which need them; "123.00" is not changed; "123456789.00" becomes "123,456,789.00".

```
dot%=idx(number$,".")                       ! Where's the decimal point?
guess
  if number$(dot%-4:dot%-4)="" then quit    ! Are there four digits left of "."?
  number$(dot%-3:dot%-4)=","                ! Stuff in a comma.
  if number$(dot%-7:dot%-7)="" then quit    ! Are there seven digits?
  number$(dot%-6:dot%-7)=","                ! Stuff in another comma.
endguess
```

The simple construct above takes care of numbers to 999,999,998.999+. [Those who program in microBASIC will note that you can insert characters in a string without overwriting any character if you use reverse value notation (string$(7:6)).]

```
ldb   ,x+
GUESS
```

We find another way to employ GUESS...ENDGUESS in assembly language. Suppose we have a character in B register and

```
  cmpb #'.
  quif eq
  cmpb #$20
  quif le
  jsr process
ADMIT
  call printerror,#badentry
  bra fini
ENDGUESS
```

will accept any entry but a decimal point, a space, a null or CONTROL, all of which are errors and all of which must end the program; other characters we process. The code at the left handles the problem.

There is, of course, an inverse form of GUESS... ENDGUESS, in which we simply invert the QUITs, and process only characters which are not a null, space, CONTROL, or period, as at left, below:

```
GUESS
  cmpb #$20
  quif hi
ADMIT
  cmpb #'.
  quif ne
ADMIT
  jsr process    ; Process only characters other than period, space, control, or
ENDGUESS         ; null.
```

So long as we remember that each QUIT trapdoors the code into the next ADMIT, the GUESS...ENDGUESS structure is easy to write and often much shorter and clearer than a series of nested IF...ELSE...ENDIFS, particularly when we write assembly language, in which the ELSEIF structure is missing.

* * *

```
if
  ...
  GUESS
  ....
  ENDGUESS
else
  GUESS
  ...
  ENDGUESS
  ...
endif
```

Not long ago, we ran into a trap in GUESS...ENDGUESS which did not crash us but certainly produced some unusual results.

As you'll see at left, we nested two GUESS...ENDGUESS structures within an IF...ELSE...ENDIF, in the position shown, in assembly language. Note there is no actual 6809 code for the "else" line. As a result, when the first GUESS...ENDGUESS was executed, the code ran us directly to the second, ignoring the "else".

We've not run into this problem in any of the other languages.

Prices, back copies, Vol. I (Postpaid), $ U.S. : Vol. I, No. 1 **not** available.

No. 2: $1.25	No. 5: $1.25	No. 8: $2.50	No. 11: $3.50	No. 14: $3.75
No. 3: $1.25	No. 6: $3.75	No. 9: $2.75	No. 12: $3.50	No. 15: $3.75
No. 4: $1.25	No. 7: $2.50	No. 10: $2.50	No. 13: $3.75	Set: $36.00

------------------------------Volume II------------------------------

No. 1: $3.75	No. 2: $3.75	No. 3: $3.75	No. 4: $3.75	No. 5: $3.75

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

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

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

Name:____________________ Disk Drive: ________ Printer:__________

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

[] Check if you're renewing; clip and mail this form with address label, please.
If you send the address label or a copy, you needn't fill in the form above.
For Canada and the U.S.: Enclose Annual Dues of $15:00 (U.S.) by check payable to ISPUG in U.S. Dollars. DUES ELSEWHERE: $25 U.S. Mail to: ISPUG, PO Box 411, Hatteras, N.C. 27943, USA. **SCHOOLS!: send check with Purchase Order.** We do not voucher or send bills.

This journal is published by the **International SuperPET Users Group** (ISPUG), a non-profit association; purpose, interchange of useful data. Offices at PO Box 411, Hatteras, N.C. 27943. Please mail all inquiries, manuscripts, and applications for membership to Dick Barnes, Editor, PO Box 411, Hatteras, N.C. 27943.
SuperPET is a trademark of Commodore Business Machines, Inc.; WordPro, that of Professional Software, Inc. Contents of this issue copyrighted by ISPUG, 1985, except as otherwise shown; excerpts may be reprinted for review or information if the source is quoted. TPUG and members of ISPUG may copy any material.
Send appropriate postpaid reply envelopes with inquiries and submissions. Canadians: enclose Canadian dimes or quarters for postage. The Gazette comes with membership in ISPUG.

ASSOCIATE EDITORS

Terry Peterson, 8628 Edgehill Court, El Cerrito, California 94530
Gary L. Ratliff, Sr., 215 Pemberton Drive, Pearl, Mississippi 39208
Stanley Brockman, 11715 West 33rd Place, Wheat Ridge, Colorado 80033
Loch H. Rose, 102 Fresh Pond Parkway, Cambridge, Massachusetts 02138
Reginald Beck, Box 16, Glen Drive, Fox Mountain, RR#2, B.C., Canada V2G 2P2
John D. Frost, 7722 Fauntleroy Way, S.W., Seattle, Washington 98136
Fred Foldvary, 1920 Cedar Street, Berkeley, California 94709

Table of Contents, Issue 5, Volume II

New Computers....................120
BEDIT gets HOST capability..........121
AND,OR, NOT integer variables....122
Report on OS9.......................123
New Pascal Compiler?.............123
Commodore Reference Book............124
Converion of Numbers.............124
Floating Point Library..............127
Floating Point Ops, Toebes.......131
CC Flags During Math................133
One Family Uses SuperPET.........134
File Transfer with Null Modem.......135
APL Express......................137
In The Twilight Zone................140
Book on Computer Repair..........142
Handling Local and Global Variables.142
Structure and Speed..............144
Bits, Bytes and Bugs................146
Fortran 77 and mFortran..........148
Fundamentals of GUESS...ENDGUESS....149

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

Bulk Rate
U.S. POSTAGE
PAID
Permit No. 47
Elizabeth City, N.C.