THE PAPER

FOR OWNERS OF THE COMMODORE PET™ PERSONAL COMPUTER

A CSD-LIPS PUBLICATION

In This Issue

Long Island PET Society Meeting

Thursday, November 13, 1980

Harborfields High School

Pulaski Rd and Taylor Ave

Greenlawn, NY

Computer Center — Rm 510

(see Back Cover for more information)

General Information

The PAPER is piublished 10 times per year by Centerbrook Software Designs and the Long Island PET Society at 98 Emily Drive, Centereach, NY 11720. Telephone (516) 585-2402.
The PAPER is mailed to subscribers during the last week of each month except June and December. Single copy price is $2 and subscription price is $15 for all 10 issues of the current volume. Subscription orders should be mailed to The PAPER, Box 524, East Setauket, NY 11733.
Third class postage is paid at East Setauket, NY 11733 (Permit # 96). POSTMASTER: Send all address changes to the address above.
The PAPER, Centerbrook Software Designs and the Long Island PET Society are in no way associated with Commodore Business Machines. CBM is not responsible for any of the contents of The PAPER unless so noted. PET and CBM are trademarks of Commodore Business Machines.
All readers are encouraged to submit articles of general interest to PET users. Mateials submitted must be free of all copyright restrictions. All contents of The PAPER are copyrighted (C) 1980 by CSD-LIPS.

Subscription Rates:

USA reresidents: $15/10 issues.
Non-USA: $20/10 issues plus $10 for airmail if desired.
No purchase orders will be accepted for subscriptions and payment must accompany all orders. Checks or money orders payable to The PAPER are acceptable. Sorry, no bank or credit cards are accepted.

Advertising:

Advertising rates will be quoted on request.

Dealers:

Any store or dealer may order at least 5 copies of each issue for retail sale. Discounts can be negotiated based on quantity and dealer commitment.

Software Exchange:

Software published in The PAPER or dsitributed through the Exchange are meant to work in the type of machine indicated. Most progarms were originally designed for the OLD ROM 8K PETs but efforts have been made to convert programs so that they will work on both ROM releases.

Staff:

Publisher: Ralph Bressler
Editor : Doug Haluza
Assoc. Ed: Roy Busdieker
Vic SantaLucia
Exchange : Charlotte Deschamps

Staff Writers: Bill Batcher
JoAnn Comito
Gerry Eisner
Jim Fowler

We're Still Here

Most of you probably know that ARESCO stopped publishing The PAPER because of Terry Laudereau's failing health. We owe a gread deal to Terry for her work on The PAPER. I'm Ralph Bressler and I will be publishing The PAPER and attempting to meet all of its obligations. Helping me will be Editor Doug Haluza.

Here on Long Island we have a large users group of about 150 members. The Long Island Pet Society (L.I.P.S.) has been publishing a newsletter for two years. This issue of The PAPER consists of the best articles from the L.I.P.S. Journal. Because many of our members are teachers there are a lot of educational articles.

We will be completing Volume 3 of The PAPER and Volume 3 of the L.I.P.S. Journal with this publication (The PAPER). We may have to print some double issues like this one to keep costs down. Completing the transfer from ARESCO to us has involved some tremendous and unexpected problems; most deal with various bureaucracies like the banks, post office and the Internal Revenue Service.

As a subscriber you can help make The PAPER work in several ways. First, encourage other users to subscribe and use The PAPER. Second, write letters to and articles for The PAPER and ask others to do the same. Third suggest that your favorite computer store, dealer or manufacturer advertise with us. Money is very tight at this time so any help will be appreciated.

The PAPER will be mailed as fast as we can produce it. Please let us know if delivery problems exist. If you have any questions or comments please feel free to write or call us.

Editor's Notes

As your Editor I work closely with Ralph preparing your newsletter. Because this is your newsletter we welcome your comments, criticism, and contributions. We cannot pay for articles, but all contributors will receive one or more software exchange credits depending upon length and content.

The PAPER is not copywritten so all articles you contribute remain your property. You may have your article republished by any of the larger paying publications. Anyone interested in republishing any article may do so by arrangment with the author; please include a note stating that the article was originally published in The PAPER.

We are especially interested in your comments on our format. Although the L.I.P.S. Journal was done in a two column format, we used The PAPER's full page format for this issue because Terry tells us you like it better that way. When we get a proportional spacing letter quality printer we will probably go back to a two column format. What do you think?

This issue of The PAPER was produced with a modified version of the CMC WPP. Future issues will be produced with WordPro 3, so if you can provide contributions on WordPro 2 or 3 disks you will get an extra software exchange credit. All disks will be returned with your software exchange programs on them.

Cross Referenced Memory Map

by Doug Haluza

To help you convert programs that worked on the old PETs so they will work on the new ROMs, I've compiled this cross referenced memory map. It exists as a composite from many different sources including the PET User's Group Newsletter, Jim Butterfield's old and new maps, Commodore's brief old and new maps, and what I've found from my own digging through disassembled copies of PET BASIC.

If you use this map to correct your own programs, you will then have a program that will work only on the new ROM PETs. It would be better to have one program that would work on both PETs. Remember that a PEEK at location 50003 tells you what ROM set you're working with, so you can write smart programs that will work with either operating system. PEEK(50003) returns a 1 with the new ROMs and a 0 with the old ones. If you want to zero the keyboard buffer and wait for a key to be pressed on either ROM set you should include this line near the beginning of your program:

```
0 P=PEEK(50003):K=525-P*367:T=135-P*82
```

Then all you would have to do to perform the operations above is:

```
100 POKE K,0 : WAIT K,7
```

OLD	NEW	DESCRIPTION
0-2	0-2	USR jump
3	14	Input device for prompt supress
4		Number of nulls after a CR (def.=0)
5		POS (also used by INPUT and PRINT)
6	15	Terminal width (unused)
7	16	Source column scanning limit
8-9	17-18	Indirect index for SYS,WAIT,POKE,GOTO
10-90	512-592	BASIC input buffer
91	4	Scan between quotes flag
92	5	Input buffer pointer; # of subscripts
93	6	Flag to remember DIMed variables
94	7	$FF=string result, 0=numeric
95	8	$80=integer result; 0=floating
96	9	Flag: list quote, DATA scan or memory
97	10	Flag: subscript or FNx
98	11	Flag: 0=INPUT, $40=GET, $98=READ
99	12	Flag: trig scan or comparison evaluation
100	64	Flag: +=normal, -=supress ouput
101		Pointer to variable psuedostack
102-103	20-21	Pointer to last temporary string
104-111	22-29	2-byte variable psuedostack
112-115	30-33	Indirect indeces
116-121	34-39	Zero page scratch pad for math functions
122-123	40-41	Pointer to start of BASIC progtams=$0401
124-125	42-43	Pointer to start of simple variables
126-127	44-45	Pointer to start of array variables
128-129	46-47	Pointer to beginning of free RAM
130-131	48-49	Pointer to bottom of strings (moves down)
132-133	50-51	Pointer to top of strings (moves down)
134-135	52-53	Pointer to top of available RAM
136-137	54-55	Current program line number

138-139	56-57	Line number for CONT
140-141	58-59	Program pointer for continue
142-143	60-61	READ DATA line number (for errors)
144-145	62-63	READ DATA pointer (initialized as $0400)
146-147	64-65	INPUT pointer
148-149	66-67	Name of current variable
150-151	68-69	Address of current variable
152-153	70-71	FOR/NEXT variable ptr./EOR,AND for WAIT
154-155	72-73	Address of current operator
156	74	Type of comparison
157-158	75-76	FNx pointer
159-161	77-79	SQR Scratch pad
162	80	Garbage collection routine constant
163-165	81-83	Jump for FNx
166-171	84-89	Floating accumulator #3 (for trig func.)
172-175	90-93	Pointer for block transfers
176-181	94-99	Floating point accumulator #1 (FACC)
176	94	Exponent + $80 (zero flag)
177	95	Mantissa MSB
178	96	Mantissa
179	97	Mantissa
180	98	Mantissa LSB
181	99	Sign of mantissa
182	100	Taylor series evaluation counter
183	101	Number of bits to shift FACC
184-189	102-107	Floating point accumulator #2 (AFAC)
190	108	FACC <-> AFAC sign comparison
191	109	Low order rounding byte for FACC
192-193	110-111	Cassette buffer length/Taylor series ptr
194-217	112-135	CHRGET routine-gets next BASIC character
200	118	CHRGOT routine - regets last character
201-202	119-120	CHRGET/CHRGOT pointer
218-222	136-140	Last random number (seed for next)
224-225	196-197	Pointer to address of beginning of line
226	198	Position of cursor on current line
227-228	199-200	G.P. pointer (tape buffer, scrolling)
229-230	201-202	Pointer to end of program- for tape write
231-232	203-204	Tape timing constants
233		Tape buffer character
234	205	Quote mode switch (0-non quote mode)
235		0=timer #1 interrupt disable
236	207	Tape write flag (EOT received)
237	208	Character read error
238	209	Number of characters in file name
239	210	Current logical file number
240	211	Current sec. address (device command)
241	212	Current device number
242	213	Maximum # of characters on current line
243-244	214-215	Pointer to start of tape buffer
245	216	Current screen line (0-24)
246	217	Buffer checksum (last key pushed)
247-248		Pointer to SAVE, LOAD and VERIFY
249-250	218-219	Pointer to filename
215	220	INST key countdown (number of INST left)
252	221	Serial bit shift word
253	222	Number of blocks remaining to write
254	223	Serial word buffer
255		Overflow for FACC to PETSCII conversions

256-266		Scratchpad for binary to PETSCII con.
267-511	256-511	6502 stack area
	256-318	Tape read error log for correction
512-514	141-143	TI and TI$ clock
515	151	Matrix coordinate of key down
516	152	Shift key status (1=down)
	153-154	Clock correction factor
517-518		1.5 jiffy clock (unused?)
519	249	Cassette #1 status switch
520	250	Cassette #2 status switch
521	155	Keyswitch PIA (STOP and RVS flags here)
522	156	Timing constant buffer
523	157	Switch: LOAD=1, VERIFY=2
524	150	Status word (ST)
525	158	Keyboard buffer pointer (# keys pressed)
526	159	Reverse video flag
527-536	623-632	Keyboard input buffer
537-538	144-145	IRQ interrupt vector(New:$E62E,Old:$E685)
539-540	146-147	BRK interrupt vector(New:$FD17,Old:$0000)
	147-148	NMI interrupt vector (New:$C389)
541	160	IEEE mode
542	161	Number of characters on current line
544-545	163-164	Cursor log (row, column)
546	165	PDB image for tape I/O
547	166	Key image
548	167	Switch: 0=cursor flash, 1=cursor off
549	168	Cursor timing coutdown (-20)
550	169	Character under cursor
551	170	Flag: 0=curosr moved, 1=blink started
552		Tape write
553-577	224-248	Screen line status table
578-587	593-602	Logical number of open files
588-597	603-612	Device number of open files
598-607	613-622	Secondary addresses of open files
608	172	Flag: 0=input from keyboard,1=from screen
265	173	X save flag
6 10	174	GPIB table length (number of open files)
611	175	Input device (0=keyboard)
612	176	Output (CMD) device (3=screen)
613	177	Tape parity
614	178	Byte received flag
616	181	Pointer in filename transfer
617-619	183	Serial bit count
621		Count of redundant tape blocks
523		Cycle counter
624		Tape write countdown
625	187	Tape buffer #1 count
626	188	Tape buffer #2 count
627	189	Tape leader counter
628	190	Write new byte/read error flag
629	191	Write start bit/read bit seq error
630	192	Pass 1 error log pointer
631	193	Pass 2 error correction ptr
632	194	0=scan, 1-15=count, $40=LOAD, $80=end
633	195	Checksum working word
634-825	634-825	Cassette #1 buffer
826-1017	826-1017	Cassette #2 buffer
	1018-1019	Monitor vector

Machine Language is Faster Than You Think

by Doug Haluza

To demonstrate just how fast machine language really is in comparison to Basic we'll try to fill all 1000 screen locations on the PET with all 256 displayable characters as fast as possible. A simple Basic program to do that might look like:

```
10 TI$="000000"
20 FOR I=0 TO 255
30 FOR J=32768 TO 33767
40 POKE J,I
50 NEXT J,I
60 PRINT
```

Try it and you'll see that it is very slow. It'll take over 16 minuets to run (removing spaces, putting it all on one line and deleting the variable reference in the NEXT statment will cut off two min.).

An equivilant machine language program might look like:

```
START:   CLD             Clear decimal mode (Precautionary)
         CLC             Clear carry flag (Precautionnary)
         SEI             Set interrupt disable (saves time)
         LDX #0          Zero X-register
         TXA             Transfer X to A (zero accumulator)
LOOP:    STA $8000,X     \         Store
         STA $8100,X       \       char.
         STA $8200,X         \       on
         STA $8300,X           \ screen
         INX              Increment X
         BNE LOOP         1000 locations done ?
         ADC #1           Increment A
         BNE LOOP         256 char. done ?
DONE:    CLI              Clear interrupt disable
         RTS              Return from subroutine
```

The first five statments are initialazation, the meat is in the loop. The four STAs store the contents of the accumulator (A) on the screen in 256 chractrer increments, so four are needed to fill the screen. Really we fill all 1024 screen memory locations, but only 1000 are displayed by the PET. The program loops through this 65,536 times in less than 2 seconds.... That's 50,000% faster!

You can try this program for yourself by using the Machine Language Monitor. Type 'M 033A 0354' at the dot, and change the output to:

```
.:  033A D8 18 78 A2 00 8A 9D 00
.:  0342 80 9D 00 81 9D 00 82 9D
.:  034A 00 83 E8 D0 F1 69 01 D0
.:  0352 ED 58 00
.   G 033A
```

'G 033A' starts the program. The RTS ($60) instruction was replaced by a BRK ($00) to return control to the monitor. See your PET manual for more information on how to use the PET TIM Monitor.

The Evolution of a Puzzle

by Bill Batcher

Don't say impossible to an intermediate student. At least, that's the lesson I learned recently while teaching BASIC to a group of fifth and sixth graders. A motivated youngster who doesn't know that something can't be done, will explore all kinds of unconventional avenues and might just arrive at a solution.

I was teaching a lesson on the string functions, LEFT$ and RIGHT$. We wrote a simple program in order to explore them:

```
10 INPUT A$
20 PRINT RIGHT$(A$,4)
30 GOTO 10
```

The students then explored what happened when they typed in their names, other words, sentences, etc. They tried including spaces, numbers and graphics. They tried words shorter than 4 letters. And they tried changing the PRINT command by changing the RIGHT$ to LEFT$ and by changing the number in parentheses. (If you are not familiar with these string functions, try experimenting with this program yourself.)

We then went a step further and made use of the LEN function, as well as the RIGHT$ and LEFT$ functions. We changed line 20 to:

```
20 PRINT RIGHT$(A$,4)LEFT$(A$,LEN(A$)-4)
```

As the pupils again experimented, they discovered the effect this simple combination had. If they input MARK JOHNSON, for example, the computer responed NSONMARK JOH.

So, I gave them a challenge. I asked them discover what you would have to problem to input for the computer to respond Jimmy Carter. Pupils approached the in various styles. Some experimented directly on the PET and others worked their answers out on paper as if they were afraid to type in a wrong answer. Many found the solution quickly, but several could not back them out of the dead end alleys their initial attempts brought them. I asked the groups that found a solution quickly to try other names like George Washington and Abraham Lincoln. (Try these on your PET.)

Once a group found a solution for JIMMY CARTER, the other names were easy to generate. I still wanted to give the 'slower' ones a chance to catch up, so I decided to stump the 'faster' groups with a puzzle I knew was impossible. I knew it was impossible, because I couldn't do it, and that was the definition I was using. But of course, I didn't tell the kids it was impossible. I just wanted to keep them busy for a while. The problem was to use the same program and input a string that would cause the computer to respond ABE LINCOLN.

While at first, this seems to be as straightforward as all the others, the groups quickly realized that here was a challenge of a different order. Again, try it yourself. There's something about that three letter name which complicates the formula which worked so easily for the other names. Of course, kids quickly realized they could solve the problem by changing line 20, so I informed them that this was not a valid solution.

Eventually, kids were calling me over and announcing that they had solved it. Of course, I found reasons not to accept each one as valid. Many left out the space after ABE or inserted a hyphen or some other character as a 'place holder', but I told them the computer had to respond with ABE LINCOLN and nothing else. Some enterprising students found ways to generate ABE LINCOLN with one or more spaces to the left of ABE. While I had to admit to myself that this was a clever solution, I was being particularly negative that day, so I insisted that ABE ahd to begin directly under the question mark, just as JIMMY had.

Well, lo and behold, before the period was, one group had called me over over over to show their solution. Sure enough, they had found a way to generate ABE LINCOLN within the parameters I had set. Later that day, I repeated the lesson for another class. Wouldn't you know, one group of students in that class also solved the puzzle. These students had discovered a different solution. Several weeks later, I presented the puzzle to a teacher's in-service class. Before the session was over a trio of teachers had solved it using a third solution.

All three solutions demonstrate some interesting characteristics of Commodore BASIC, characteristics that may have some other practical applications.

Watch for the three solutions and any other I collect in an upcoming issue. If you find any solutions please let me know!!

When 100 < 99

by Ralph Bressler

I had occasion to want to treat numbers, specifically, student grades as strings. I wanted to set up a general sort routine which would sort a set of data based on any field in the record. So, I could alphabetize by name or rank order the students by final average. The sort I worked out was fast but it would work best if all the fields were strings rather than numeric.

I set up the sort and everything went fine. The routine would sort on the basis of any key field and it was FAST ! One problem developed which made things less than perfect. I have one very smart student who happened to have a 100 test average. When the students were sorted with tests as the key field this student always went dead LAST. I finally found that the line in my program which compared the test averages in string form was to blame. A quick check showed that the computer considered 100 in string form LESS THAN 99 in string form and, in fact, less than everything down to 11. It did show that 100 was greater than 10. I could not figure this out. I was stumped ! With the help of JoAnn Comito I now understand the reason. Do you know why ? Actually after she explained it the answer seemed rather obvious. Try the following on your PET and see if you know why:

```
    IF STR$(100)<STR$(99) THEN ?"OK
    IF STR$(100)<STR$(11) THEN ?"OK
    IF STR$(100)<STR$(10) THEN ?""OK
    IF STR$(100)<STR$(9) THEN ?"OK
```

The reason, of course, is that the comparisons are based on the ASCII values of each character in the string. The comparison starts at the left most character in the string and continues until a difference occurs. At this point one string is the 'winner' and is judged less than or greater than the other. Null characters have no value and are, therefore, less than anything else.

Machine Language is Still Faster than you Think

by Doug Haluza

Using a bubble sort to sort a list of numbers will show machine language's blinding speed.

First some background on how a bubble sort works. A series of passes are made through the data. During each pass each element is compared to the next and is swapped if it's larger (i.e. if they're backwards). A flag is set each time a swap is made and checked after each pass. If no swaps were made the pass is complete. It's called a bubble sort because the smaller numbers 'bubble up' to the top (Actually if the table is arranged in ascending order the larger numbers sink to the bottom forcing the smaller ones up).

A small sort might look like:

```
  ** *     *       *
4 1111    1111    1111    1111
1 4333    3333    3222    2222
3  444     422     333     333
5   52      44      44      44
2    5       5       5       5
 PASS-1 PASS-2 PASS-3 PASS-4
*=Swap was made
```

First the 4 is compared to the 1;since they're backwards they're swapped. The 4 is then compared to the 3 and again they're swapped. When the 4 is compared to the 5 no swap is necicary, but 5 is greater than 2, so they're swapped.

This procedure continues for each pass until no swaps are made and the sort is then complete. One way to sort 100 numbers between 0 and 100 (e.g. test grades) would be:

```
100 F=0:FOR I=0 TO 98:IF A(I)> A(I+1) THEN H=A(I+1):A(I):A(I)=
    A(I+1)=H:F=1
110 NEXT:IF F THEN 100

A(0)-A(99)=Numbers to be sorted
H=Holding variable
F=Swap made flag
I=General purpose counter
```

An equivilant assembly language routine is shown in listing-1. Its efficiency is increased somewhat by sorting from the bottom up rather than from the top down as in the BASIC routine.

In the assembled routine in Listing-1 that is also used in Listing-2, the table of numbers to be sorted was set up to reside in the first, and part of the second cassette buffers. Because the routine itself resides in the upper part of the second cassette buffer, the table may have as many as 255 entries. Naturally the table may be moved anywhere else in RAM by changing the 5 table address references.

TABL,X is the table. It's used like A(I) with the X-register being used as the pointer instead of I. F, the flag in the BASIC routine, is replaced by the Y-register. Finally instead of using a holding variable H one of the items is pushed on the stack and pulled later.

Both routines can be used to sort up to 256 numbers (the BASIC routine can handle more if you have the new ROM's). With the machine language routine, however, the numbers must be between 0 and 255. It can be changed to handle any combination of more, larger, and/or negative numbers by using multiple precision techniques, but they are beyond the scope of this discussion.

The BASIC program in listing-2 will sort up to 255 numbers in both BASIC and machine language, print out the sorted numbers, the time for each sort, and the median.

Depending on the number of items to be sorted, machine language can be almost 60,000 faster than BASIC. For this reason the machine language routine should be very useful. Feel free to modify and adapt it to any of your applications.

Listing-1 Assembly language routine

```
STRT LDY #0         Zero swap made flag
     LDX #99        Set counter for 100 numbers
GO   LDA TABL,X     Get one item
     CMP TABL-1,X   and compare it to the one before
     BCS NOSW       If it's not smaller no swap's needed
SWAP PHA            otherwise save it on the stack
     LDA TABL-1,X   Load the other
     STA TABL,X     and switch it
     PLA            Retrieve the first one
     STA TABL-1,X   and switch it
     LDY #-1        Set swap made flag
NOSW DEX            Decrement counter
     BNE GO         Done all 99 compares?
     TYA            If so check swap flag
     BNE STRT       If a swap was made do it again
     RTS            Otherwise sort is finished
```

Listing 2 BASIC Demo Program

```
1 REM SORTING PROGRAM FOR NEW AND OLD PETS BY DOUG HALUZA
2 REM LINES 10-30 SET UP THE MACHINE LANGUAGE PROGRAM. SYS 900 CALLS IT.
3 REM THE NUMBERS ARE STORED IN A(N) AND MEMORY LOCATIONS 634 TO 634+N-1
4 REM THE RANDOM SORT OPTION SETS UP A TABLE OF RANDOM NUMBERS TO BE
5 REM SORTED. THE WORST CASE OPTION GIVES THE MAXIMUM SORT TIME BY SETTI
6 REM UP A TABLE OF NUMBERS IN DESCENDING ORDER (E.G. 100,99,98...2,1).
10 DATA160,0,162,99,189,122,2,221,121,2,176,13,72,189,121,2
20 DATA157,122,2,104,157,121,2,160,255,202,208,232,152,208,225,96
30 FORI=900TO931:READN:POKEI,N:NEXT
40 INPUT"NUMBER TO SORT";N:DIMA(N-1):POKE903,N-1
43 DEFFNN(N)=N-I:INPUT"RANDOM OR WORST CASE SORT";A$
47 IFA$="R"THENDEFFNN(N)=INT(RND(1)*100+1)
50 FORI=0TON-1:A=FNN(N):POKE634+I,A:A(I)=A:NEXT
60 PRINT"JIFFIES","HHMMSS":TI$="000000":SYS900:PRINTTI,TI$,"MACH LANG"
99 TI$="000000"
100 F=0:FORI=0TON-2:IFA(I)>A(I+1)THENH=A(I):A(I)=A(I+1):A(I+1)=H:F=1
110 NEXT:IFFTHEN100
120 PRINTTI,TI$,"BASIC"
130 PRINT"BASIC MEDIAN"(A(N/2)+A(N/2-.5))/2
140 PRINT"MACH. MEDIAN";:I=N/2+633.5:PRINT(PEEK(I+.5))/2
200 FORI=634TO633+N:PRINTLEFT$(STR$(PEEK(I))+"   ",4);:NEXT:PRINT
```

Stringing Your PET Along

by JoAnn Comito

All of you are probably familiar with numeric constants (1.2, -3, 1E6,...), numeric variables (A, R2, D(2),...) and numeric functions (SQR, TAN, ABS,...) in BASIC. Some of you may not be as familiar with string constants, variables, and functions in PET BASIC.

What is a string? Anything you type in from the PET keyboard and enclose between quotes is a string. Letters, numbers, graphic characters, even cursor controls can be included in strings. (The exception to this is the quote itself, since it serves as the string delimiter.) Here are a few examples of strings: 'HELLO'; '1.24'; ' cd cd cd HELLO cd cd cd 123'. If you haven't worked with PET strings before, try typing the last example on your PET preceded by PRINT. When you enter the string from the keyboard, the 'cursors down' will appear as Q's in reverse field. But when the PET actually prints the string it will move the cursor down three lines, print HELLO, and go down three lines before printing the 123.

In the same way that numeric variable names can be used to represent numbers, string variable names can be used to represent strings. String variables have the same format as numeric variables, but are followed by a $. For example: A$, R2$, D$(2). String variables can be used everywhere string constants are used in a program. In LET, IF, PRINT, INPUT, etc. statements. For example:

```
10 A$="HELLO"
20 INPUT B$
30 C$="SAME"
40 IF A$=B$ THEN PRINT C$
50 IF A$<>B$ THEN PRINT "NOT "C$
```

BASIC contains a number of functions that allow us to manipulate strings. These functions include LEFT$, RIGHT$, MID$, and LEN.

LEFT$("HELLO",2) would return the string "HE", the two leftmost characters of the string 'HELLO'. This function requires two arguements, the first is a string, the second a numeric expression. The arguements of the LEFT$ function can also be variables. Try the following progam:

```
10 A$="ABCDEFGH"
20 FOR I = 1 TO 8
30 B$ = LEFT$(A$,I)
40 PRINT B$
50 NEXT I
```

RIGHT$("HELLO",2) would return the string "LO". As you probably guessed RIGHT$ does the same thing as LEFT$ except starting from the right. Try the program using RIGHT$ instead of LEFT$ in line 30.

The MID$ function requires two or three arguements. The first arguement specifies the string to be examined, the second specifies a staerting location along that string, and the third specifies the number of characters to return. If the third parameter is left out it returns the rest of the string. For example: MID$("HELLO",3,2) would return the string "LL".

The function LEN simply returns the number of characters in a specified string. LEN("HELLO")=5.

If you wanted to scan a word, or group of words and find out how many times the letter 'E' was used, you would use MID$ and LEN functions. Try the following:

```
5 N=0
10 INPUT B$
20 FOR I=1 TO LEN(B$)
30 IF MID$(B$,I,1)="E" THEN N=N+1
40 NEXT I
50 PRINT"THERE ARE"N"E's IN"B$
```

The string functions are not only useful for examining and manipulating strings, but can be used as part of a simple routine to add cursor control to your programs. The following routine will allow you to specify two numbers, D and R, for the number of lines down and spaces over, you wish to position the cursor from the home position. At the beginning of your program define the two strings D$ and R$ as:

```
10 D$="(h)(25 down)"
20 R$="(40 right)"
 .
1000 REM THIS IS THE SUBROUTINE
1010 P$=LEFT$(D$,D)+LEFT$(R$,R)
1020 PRINT P$;
1030 RETURN
```

Suppose that you wanted to print something in the fifth row from the top of the screen, starting in the tenth column. You would include the following statement in your program:

```
D=5: R=10: GOSUB 1000: PRINT"....
```

When PET prints P$ in line 1020, it does not print any characters on the screen, but it does home the cursor then print the required number of down and right cursors. The semicolon after P$ in line 1020 keeps the cursor at that position ready to print anything you want.

More Strings Attached

by JoAnn Comito

The problem I would like to discuss arose in a program designed to give students practice manipulating algebraic expresions. The response that the students are to enter is an algebraic expresion in two variables and includes three terms. For example: 3X+5XY-2Y. Since there are many equivalent correct answers, the problem is getting the program to recognize all the correct variations. There are two approaches to this problem.

One approach is to structure the exercise in such a way that the student has only one correct option for each entry. For example, the student could be directed to enter the coeeficient of the X term, followed by the coefficient of the XY term, etc. The program would only have to check the three coefficients. While this approach eases the burden on the programmer, it also creates a very artificial situation for the student. If the student were doing these exercises with a paper and pencil for the teacher to correct, the student would be required to determine not only the correct coefficients, but the correct terms as well. On the other hand, the teacher would have no difficulty recognizing all the correct variations of the expected response. The student could write: 5XY+3X-2Y; 5XY-2Y+3X; 3X + 5XY - 2Y; etc. and they would all be judged correct by the teacher, but incorrect by the computer.

The second approach involves some work on the part of the programmer. The goal is to enable the computer to recognize a wide variety of correct responses. The student is given a minimum of prompting and must enter not only the coefficients, but the correct varoables as well. The student is also allowed maximum flexibility with respect to the format of the response.

Before continuing, it should be made clear that two strings are equal only if they are identical character for character. While 4, 2+2, 2↑2, 5-1, etc. would all be recognized as equivalent numeric expressions, '4', '2+2', etc. would not be considered equivalent strings. Neither would 'HELLO' and 'HE LLO' be considered equal (spaces count!).

Before being able to check the student's response, all extraneous characters would have to be deleted from it. By extraneous characters I mean characters that are not needed, but that don't necessarily make the response incorrect. For example, spaces, multiplication symbols, parentheses, etc. should all be deleted from the student's response. The following routine will accomplish that task.

Suppose A$ represents the student's response.

```
100 REM DELETES " ","*","(",")"
110 L=LEN(A$)                     (determines length of A$)
120 FOR I=1 TO L
130 B$=MID$(A$,I,1)               (examines 1 character of A$ on
                                   each pass through loop)
140 IF B$<>" " AND B$<>"*" AND    (if B$ is not extraneous
    B$<>"(" AND B$<>")" THEN190    then continue with next character)
150 TL$=LEFT$(A$,I-1)             (TL$ stores all characters form the
                                   beginning of A$ up to, but not
                                   including the extraneous character)
160 TR$=RIGHT$(A$,L-I)            (all charcaters from the end of A$
                                   up to, but not including the
                                   extraneous charcater are in TR$)
170 A$=TL$+TR$                    (A$ is redefined as the
                                   concatenation of TL$ and TR$)
180 I=I-1                         (all characters after the deleted
                                   character are moved back one
                                   space to the left, counter is
                                   decremented to align it properly)
190 NEXT I
```

By the time this is completed, all extraneous charcters will have been deletd from the student's response. The only valid characters that should be left in A$ are '+', '-', 'X', 'Y' and some numbers. In order for A$ to be a correct response, it should contain an X, Y, and XY term, each multiplied by the correct coefficient. Another routine is needed to scan A$, find the variables and check the coefficients.

While this routine is designed to analyze a specific algebraic expresion, it can be modified to recognize other expressions as well. Compensations would have to be made for the number of terms in the expression and the variables that are to be found in each term.

The routine, with explanations follows. Assume that the student's response is stored in A$ and that all extraneous characters have been removed. Assume further that the three required coefficients are stored in an array C, where C(1) stores the correct value of the X coefficient; C(2) stores the correct Y coefficient and the correct XY coefficient is stored in C(3).

```
100 REM EXPRESSION ANALYZING STRING ROUTINE
110 T=0                           (T stores the number of terms found)
120 IFLEFT$(A$,1)="+"OR LEFT$     (checks first character of A$, if
    (A$,1)="-"THEN140              '+' or '-' then OK)
130 A$="+" + A$                   (+ is added to the front of A$)
140 A$=A$ + "+"                   (+ added to end of A$)
150 GOSUB 200                     (scanning routine)
160 IF T>3 THEN PRINT "WRONG":    (makes sure only 3 terms found, else
    RETURN                         A$ wrong, return to main program)
170 FOR I=1 TO 3: IF C(I)<>0      (makes sure all 3 terms found and
    THEN PRINT"WRONG":RETURN       returns to main program)
180 NEXT I
190 PRINT"RIGHT":RETURN           (if two tests passed, A$ correct)
200 FOR I=1 TO LEN(A$)            (start scanning routine)
210 B$=MID$(A$,I,1)               (store one charcater fo A$ in B$)
220 IF(B$="+" or B$="-" AND       (if term flag (TF) is not set to 1
    TF=0 THEN TF=1:XF=0:YF=0:      and a '+' or '-' found then start
    NEXTI                          of term --TF set and Xflag and
                                   Yflag initialized--continue)
230 IF (B$="+" or B$="-")         (if TF set to 1 and '+' or '-' found
     AND TF=1 THEN GOSUB 400:      end of term, check coefficient-
     GOTO 200                      scan A$ from beginning)
240 IFB$="X" THEN XF=1            (X found, set Xflag to 1, delete X,
    GOSUB300:NEXTI                continue scan)
250 IF B$="Y" THEN YF=2:          (Y found, set Yflag to 1, delete Y,
    GOSUB300:NEXTI                continue scan)
255 IF (B$>="0" AND B$<="9")      (now character should be 0 to 9 or
    OR B$="." THEN 260             '.', if OK get next character.)
257 RETURN                        (wrong charcater-return to 160.)
260 NEXT I
270 RETURN                        (all characters done- return to 160)
300 REM DELETE ROUTINE
310 TL$=LEFT$(A$,I-1)
320 TR$=RIGHT$(A$,(LEN(A$)-1))
330 A$=TL$ + TR$                  (A$ redefined with X and Y deleted)
340 I=I-1                         (realign pointer)
350 RETURN                        (go back to 240 or 250
400 REM CHECK COEFFICIENT
405 T=T+1                         (increment # terms found by 1)
410 TF=0                          (term ended--reset TF to 0)
420 S=XF + YF                     (S sum of X and Y flags)
425 IF MID$ (A$,2,1)=B$ THEN      (if 2nd character of A$ is '+' or '-'
    C=1:GOTO440                    then coefficient is 1)
430 C=VAL(A$)                     (returns numeric part of A$)
440 IFC(S)=C THEN C(S)=0          (C compared with value stored in
                                   coefficient array if correct, cell
                                   in array set to 0 to indicate use)
450 A$=RIGHT$(A$,(LEN(A$)-I+1))   (delete leading coefficient)
                                   coefficient)
460 RETURN                        (go back to 230 to scan new A$)
```

Assembly Language Programming

I. An Introduction

by James Fowler

This is a series of articles for the programmer who works well enough in BASIC to get done what needs to be done but who suspects the PET is capable of something more (and it certainly is).

The 'heart' of the PET is the microprocessor. It is a network of switches and conductors so arranged that the switches are set by electrical signals (the 0's and 1's in an incoming word) and these settings affect the next incoming word. Various kinds of settings correspond to logical 'AND', or arithmetic additions and so forth. The 'vocabulary' of words that set the processor and the 'grammar' of how the words and the data are combined constitute the MACHINE LANGUAGE of the microprocessor.

Humans find machine language almost impossible to read and very difficult to write. Of course, one mistake can be catastrophic. So computer engineers make a second language available which is more human-oriented. This is ASSEMBLY LANGUAGE. It bears a one-to-one relation with machine language. Many computers are supplied with a program called an ASSEMBLER that takes assembly language as input and turns out the equivalent machine language as output. There are assemblers available for the PET but they take a lot of memory. We will start without one.

Why bother with assembly language at all ? The PET does a good job with BASIC and even points out many kinds of errors for you. In assembly language programming you will have to do this all for yourself. It's a lot of work; debugging can be very frustrating; so why bother ? There are three reasons: 1) You can do things in assembly language that you can't do in BASIC. 2) Assembly programs generally use a lot less memory and space than BASIC does to do the same thing. 3) Assembly programs run much faster. Here are two examples:

This is a program to draw a white frame on the screen. The main part of the program (all except line 30) takes 122 bytes. It takes almost 1 second (actually 9/10) to run. (The last line is just to suppress the ready until you hit the STOP.):

```
10 PRINT CHR$(147):FORI=32768 TO32807:POKEI,160:NEXT
15 FORI=32847TO33727STEP40: POKEI,160:NEXT
20 FORI=33726TO33688STEP-1: POKEI,160:NEXT
25 FORI=33648TO32808STEP-40: POKEI,160:NEXT
30 GETX$:IFX$=""THEN55
```

Here is another program. Line 15 reads 66 bytes of machine language into memory. Then the program waits (line 30) for you to hit SPACE. Then it executes the machine program it put in memory and waits for STOP. You can run the machine program without the BASIC program (once it has loaded memory by running at least once) with command SYS 2560.

```
10 A=2559:FORI=1TO66
15 READX:POKE(A+I),X:NEXT
20 PRINTCHR$(147)"hit space"
25 GETA$:IFA$=""THEN25
30 IFA$=" "THEN SYS2560:GOTO25
```

```
35 IFA$=CHR$(13)THEN END
40 DATA169,128,133,2,169 ,0,133,1,170,168,169
50 DATA160,145,1,200,192,40 ,208,249,136,24,165
60 DATA1,105,40,133,1,144 ,2,230,2,169,160,145
70 DATA1,232,224,24,208 ,236,145,1,136,16
80 DATA251,202,200,56,165 ,1,233,40,133,1,176
90 DATA2,198,2,169,160,145 ,1,202,208,238,96
```

If it doesn't run right you probably made an error in one of the DATA lines.

The machine program took up about half the space (66 bytes) of the BASIC program (122 bytes). Yet it executed so fast you couldn't time it. I calculated it took about one thousandth of a second to run - 1000 times faster!

How do you get started in machine language ? Get a good book - study - practice. I used Caxton Foster's book 'Programming a Microcomputer: 6502' published by Addison-Wesley. It was one of the first on 6502 Assembly Language which is what the PET 'speaks'. Now there are many such books - look them over and take the one with the style you like. When you get serious you will need the reference the professionals use: '65XX Programming Manual' from MOS Technology.

Next installment: How to handle HEX, load and run.

(Ed's note: 6502 Assembly Language Programming by Leventhal and published by Osborne is highly recommended. A similar book published by Sybex has many errors and is confusing. Many books are now available but only the Abacus Software Machine Language Guide directs itself to the PET. Watch for a review of these books in an upcoming issue.)

Poke a Border

by Gerry Eisner

The PET screen contains 40 columns from left to right and 24 rows from top to bottom. Each space has a number associated with it. At the top left this number is 32768; beside it is 32769 and below it is 32808 (32768+40). The lower right hand corner is 33767; this is the last space on the screen. These numbers are really memory locations and we may change them by using the POKE command. Normally a blank screen would have a 32 in each of these locations since 32 is the PET code for a blank space. To see how this works try:

```
POKE32768,1:POKE32769,2:POKE32808,3
```

POKE x,y is the command to put the character represented by the number y in the position given by x. In this case 1 is the code for A; 2 is for B and 3 is for C. There are 255 possible characters so y can be from 0 to 255. There are 1000 possible spaces so x can be from 32768 to 33767. To find the code for a character try this:

1) Home the cursor and type the charcter in the HOME position. Hit RETURN.
2) Type ?PEEK(32768) and hit RETURN. The number you get is the code for your character. Call this number y.
3) Clear the screen, hit RETURN and type POKE32768,y. You should get your character in the HOME position.

POKE changes the contents of a memory location. PEEK shows you what is in a memory location. The character codes for the PET are unique to this machine and are not standard. To fill the top row you could:

```
      10 FORI=1TO40
      20 POKE32767+I,83
      30 NEXTI
```

Here we have made a loop to repeat the POKE instruction 420 times. Each time through the loop the location gets bigger by 1. To fill the bottom row along with the top row we could add:

```
      25 POKE32767+24*40-I,83
```

This addition places a second POKE location 24 rows from the top. To add the side borders:

```
      40 FORJ=32767+41TO32767+24*40 STEP40
      50 POKEJ,83:POKEJ+39,83
      60 NEXTJ
```

To obtain the left side border you started at the 32808 location and POKEd every 40 spaces which is the purpose of the STEP command. The second POKE gives the same image 39 spaces to the right or at the right border.

To polish up this program, add a CLEAR screen the beginning and a loop at at the end to prevent the READY message and the scrolling of the screen.

A further refinement on POKing borders is to nest them with alternating symbols. For instance, if you want to create a second line under the top row in our example, try:

```
      5 E=83
      10 FORN=0TO1
      20 FORI=1TO40-2*N
      30 POKE32767+41*N+I,E
      40 NEXT I
      50 E=169-E
      60 NEXTN
```

N sets the number of rows. E alternates between 83 (hearts) and 86 (large X). Try other numbers. For example, 160 is a solid block and 32 is the empty space.

Please note that we have used an I loop working within an N loop. We positioned the row of hearts first, then picked up the X symbol, back to a new number for N, and finally placed the row of X's. The POKE starting point changed when the N changed to 1 and the number of POKEs was reduced at the same time.

The following is a routine to accomplish nested borders with n rows of alternating hearts and large X's.

```
      5 PRINT"(clr)".
      10 A=32767:B=33727:C=32808:D=83
      20 FORN=0TO10
      30 FORI=1TO40-2*N:POKEA+41*N+I,E:POKEB-39*N+I,E:NEXTI
      40 FORI=c+41*NTOB-39*N STEP 40:POKEI,E:POKEI+39-2*N,E:NEXTI
      50 E=169-E
      60 NEXTN
      70 GOTO70
```

Moving Around the Screen

by Ralph Bressler

Many times programs require moving a 'man' around the screen which often uses the POKE command. The standard is to use the 2 key for down, 4 for left and so on. To use all 9 keys on the number pad you would have to use 9 IF..THEN statements to check the input and then add or subtract from the present screen location to move the 'man'. For example:

```
270 IF A=2 THEN YP=YP+40
280 IF A=4 THEN YP=YP-1
290 IF A=6 THEN YP=YP+1
300 IF A=8 THEN YP=YP-40
```

Of course, you need 4 more statements for the diagonals and one for '5' which usually stops movement. Now this doesn't move the 'man', it only indicates which way he should go. A is the number from the keypad which is usually input using the GET statement. YP is your position on the screen from 32768 to 33767. To actually move the 'man' you would have to include a line like:

```
310 POKE YP,87
```

This will not erase the old 'man' nor will it check to see where the man is moving. Doug Haluza gave me a routine which does this whole thing much quicker and it is included in the program below.

```
 10 PRINT"(clr)":CLR
20 FP=33268:YP=FP
30 POKE YP,42 : REM 42 IS *
40 GETA$:IFA$=""THEN40
50 A=VAL(A$):IF A<0 OR A>9 THEN 40
60 POKE YP,32: REM 32 IS BLANK
70 YP=YP+SGN(INT(3*(((A-1)/3)-INT((A-1)/3))-.9))+
   40*SGN(INT(A/3-1.1))
80 IF YP<32768 OR YP>33767 THEN YP=FP
90 POKE YP,42
100 GOTO40
```

This routine is shorter and really does perform its stated function. It also provides a challenge! Try to decipher how and why line 70 works.

PET Tutor Updates

Lesson 1 calls for information to be POKEd onto the screen for user viewing and manipulation. On the old ROM PETs the 'image of screen memory' was 4K wide. The new ROM PETs use the upper 2K for memory expansion. The program uses the section of RAM now assigned to memory expansion. The cure is to set the address to a lower value used as screen memory by BOTH machines.

The following changes are necessary:

```
Line 8285: POKE 36000 becomes POKE 32928
Line 8680: POKE 36160 becomes POKE 33088
```

Not Ifs Ands Or Buts

by JoAnn Comito

The If...THEN statement is one of the most powerful statements in the BASIC language. It enables you to determine if certain conditions have been met or not. It is a decision making statement, performing certain operations if a condition is satisfied, performing other operations if the condition is not satisfied. The format of the IF...THEN statement is as follows:

```
    n If [logical expression] THEN [BASIC statement]

    n = line number
     logical expression = discussed in text
    BASIC statement = any valid BASIC expression
```

When an IF...THEN statement is encountered in your program, the logical expresion is evaluated by the PET. If the expresion is true, then the BASIC statement following the word THEN is carried out. If the logical expression is false, then whatever follows the THEN is ignored and the next numbered line of the program is executed.

What kind of BASIC statement can follow the THEN? Any BASIC statement can follow the THEN: PRINT, LET, GOTO, IF...THEN, ON GOTO, etc. Some statements such as DATA will be ignored and others, like another IF...THEN become unweildy.

What is a logical expression and how do you know if it is true or false? The second part of the question helps to answer the first part. A logical expression is an expression whose truth or falsity can be determined. The expression 4+2 is true to the PET, but it is not really a logical expression. on the other hand, we can determine the truth or falsity of the expression 4+2=7. In this case we have a logical expression whose value is false. Generally, in a simple logical expression, two items are compared and a statement is made relating them. The relation between the items can be expressed in different ways. They might be the same (=), they might be different (<>), the first might be larger than the second (>), or the other way around (<). The following are examples of simple logical expressions: A+2<=7; A=B; B$<>"HELLO"; A>4. They are all logical expressions, because in each case, assuming we know the value of the variables, we can determine whether or not the expression is true. They are simple logical expresions, because each statement contains only one comparison.

We must also consider compound logical expressions. These are composed of two or more simple logical expressions connected by AND's and/or OR's. For example: 2+4=6 AND 3<5; C=A+B OR B$="PET" AND X<=E; (A=1 OR A=2) AND (B=3 OR C=17).

While the truth or falsity of simple logical expresions is easily determined, how can we determine the truth value of compound logical expressions? The method is similar to that for evaluating arithmetic expressions. If you were to evaluate the following: 2*3+(1+4*5), you would not attempt to do it all in one shot. You would group the numbers so that you evaluated one operation at a time according to the 'order of operations' rules: first parentheses, then multiplication, then addition. For example:

```
2 * 3 + (1 + 4 * 5)

  6   + (1 +  20)

  6   +   21

     27
```

When evaluating compound logical expressions, first determine the truth value (truth or falsity) of each simple expression. Then think of OR as addition and AND as multiplication. In other words, first you evaluate expressions in parrntheses, then AND's, then OR's. Here are the 'addition' and 'multiplication' tables for OR and AND (where T stands for true and F stands for false).

```
T OR T = T                    T AND T = T
T OR F = T     OR             T AND F = F     AND
F OR T = T   TABLE            F AND T = F   TABLE
F OR F = F                    F AND F = F
```

If either one or both of the simple expressions in an OR statement are true, then the compound expression is true. The compound expression is false only when both simple expressions are false. AN AND statement is true only when both of the simple expressions are true. If either or both of the simple expressions is false, then the compound expression is false.

Let's evaluate the following expression:

```
4 + 6 = 2 AND 5 + 7 < 10 OR (2 > 3 OR 4 > 1 AND 5 < 6)

   F     AND      F     OR (  F    OR    T    AND    T   )

         F             OR (  F    OR          T )

         F             OR              T

                       T
```

therefore the expression is true.

Of course you do not have to evaluate the logical expression when you use in IF...THEN statements in your program. PET will do all the evaluating. However, you must know how to evaluate these expessions so that you can be sure PET decides things the way you want them decided. If we take the same expression as above and add a pair of parentheses, we get a different result:

```
4 + 6 = 2 AND (5 + 7 < 10 OR (2 > 3 OR 4 > 1 AND 5 < 6))

    F     AND (      F    OR (  F    OR    T    AND    T   ))

    F     AND (      F    OR (  F    OR         T          ))

    F     AND (      F    OR          T )

    F     AND             T

           F
```

therefore, the expresion is false. You must be sure to include parentheses where they are needed in order to express what you mean, otherwise you will have surprising results.

When would you use compound logical expressions in an IF...THEN statement? Here are a couple of examples:

1) You want to be sure the user input an integer, I, between 1 and 10.

```
   100 IF I<>INT(I) OR I<1 OR I>10 THEN PRINT "TRY AGAIN"
       BETWEEN 1 AND 10"
```

2) You are looking for animals, A$, for your zoo. You will take a panda of either sex, S$, but you are only interested in a lion if it is a female.

```
   200 IF A$="PANDA" OR A$="LION" AND S$="FEMALE" THEN PRINT"OK"
```

The last component of the logical expression to discuss is NOT. NOT simply reverses the truth value of the expression immediately following it. NOT 4+2=7 is true. NOT 4+2=6 is false. NOT (4+2=6 AND 3<=4) is false. NOT (4+2=7 AND 3<=4) is true.

You must be careful when an IF...THEN statement is the first statement in a multiple line statement line. If the logical expression is false, then the program continues on the next numbered line. The later statements in the line may never be executed. For example:

```
     10 IF A=3 THEN 100: PRINT"A DOES NOT EQUAL THREE"
     20 PRINT"THIS IS LINE TWENTY"
```

If A does equal three, then the program will continue at line 100. If A does not equal three, then the program will continue at line 20. In either case the second part of line 10 is never encountered.

And now, for something completely different. This routine will move a ball around the screen using the number pad. It avoids the 'snow' you get when using POKEs and you need only determine if an even or odd number was typed (line 130). If you are only using 2, 4, 6, and 8 then you can omit line 130 and O$.

MOVE A BALL

```
 5 PRINT"[illegible]";
 10 O$="[illegible]" : E$="[illegible]"
 100 GETA$:IFA$=""ORA$<"1" OR A$>"9" THEN 100
 110 PRINT"▌▌ ";
 120 A=VAL(A$)
 130 IF INT(A/2)<>A/2 THEN PRINT MID$ (O$,(3*A-1)/2,3);:GOTO150
 140 PRINT MID$(E$,(A-1),2);
 150 PRINT"●";
 160 GOTO100
```

BASIC Does It Better

by Ralph Bressler

About a year ago the 16/32K PETs began their trek across the country. These machines contained a slightly altered Microsoft BASIC, a new character generator and wholesale changes in the page 0, page 1, and page 2 memory locations. Now the retrofit ROMs are available for the old 8Ks and all new PETs will have the new ROMs. The new ROMs take care of some serious mistakes in the old ones and are worth having. However, many of us have good programs which ran well on the old machines but will not on the new. Certainly, anything written in machine language will have to be almost completely overhauled. Programs written in straight BASIC will need no changes except for the reversal of upper and lower case. Software that uses POKE statements may have to be revised.

The point of this article is that, as much as possible, software should use only standard BASIC and avoid refering to memory locations which may change whenever the manufacturer likes. The following examples show how you can use BASIC to do some of the work that POKEs do. For those of you just starting these techniques should also be of interest.

POSITIONING A POINT ON THE SCREEN

The following lines can be used to position a point 10 lines down and 20 spaces over.

```
Old ROMs: 100 POKE 245,9 : PRINT : POKE 226,10 : PRINT"*"

New ROMs: 100 POKE 216,9 : PRINT : POKE 198,10 : PRINT"*"

BASIC   : 5 DD$="(home)(25 down)"
          100 PRINT LEFT$(DD$,10) : PRINT TAB(10) "*"
```

(Note: Anything in parentheses is not to be typed as shown. For example, (home) means hit the HOME key once.)

CLEARING THE KEYBOARD BUFFER

Good programs make the user hit as few keys as possible to get the desired result. This means that yes/no questions or those that require only a one character response should use GET statements. For example,

```
5 PRINT"Do you want instructions"
15 IFA$="Y" THEN GOSUB 1000 @ INSTRUCTIONS
10 GETA$:IFA$=""THEN10
20 REM Long set of calculations
80 PRINT"Another run"
90 GETB$:IFB$=""THEN90
100 IFB$="Y" THEN 20
```

In this case the computer does a long set of calculations and then asks if the user wants to do it again. If the user is impatient and presses a key when the computer is calculating, the GET statement in line 90 will use that character. This means the program may end or perform some other undesired result. For naive users it is best to

CLEAR the KEYBOARD BUFFER before each input. The program would then look like this.

```
5 PRINT"Do you want instructions"
10 GETA$:IFA$=""THEN10
15 IFA$="Y" THEN GOSUB 1000 @ INSTRUCTIONS
20 REM Long set of calculations
80 REM Clear the buffer here
85 PRINT "Another run"
90 GETB$:IFB$=""THEN90
100 IFB$="Y"THEN20
```

Now for line 80 in the last case we may use:

```
Old ROMs: 80 POKE525,0

New ROMs: POKE158,0

BASIC   : 80 FORI=1TO9:GETA$:NEXT
```

STOP KEY DISABLE

Again, a naive user can find any number of ways to destroy your programs. They may, through no fault of their own, press the stop key and break the program. Not knowing that a simple CONT will get them back where they were, they often walk away. In most educational and demonstration programs making the STOP key inoperative makes good sense. Unfortunately, there is no standard BASIC way to do this. We must therefore use two different POKE locations or make our program smart so that it knows what machine it is running on. A smart program will need only one statement to disable the stop key.

```
Old ROMs: POKE 537,136 : REM disables Stop
          POKE 537,133 : REM Enables Stop

New ROMs: POKE144,46  : REM Disables Stop
          POKE144,43  : REM Enables Stop
```

SMART:

On the old ROMs PEEKing at BASIC results in a value of 0 since BASIC was protected. With the new ROMs a PEEK(50003) will return a 1. Len Lindsay first publicized this technique in the July 1979 issue of Microcomputing and we are indebted to him for the idea. For a complete explanation and other examples see that article.

Notice in the above example that the POKE locations between old and new differ by 393 and the values for disable and enable are 3 different in each case. Therefore a a short program like the following will work for both sets of ROMs.

```
     1 PT=PEEK(50003) : PL=537-PT*393 : POKE PL,PEEK(PL)+3
```

Since PT (PET type) will be 0 on the old ROMs, PL will be 537. On the new ROMs PT will be 1 and PL will be 144. We POKE into these locations whatever was there, PEEK (PL), plus 3. To enable this key simply POKE PL, PEEK(PL)-3. Caution must be used here since these statements can obviously hang up the PET.

SCREEN MEMORY

The screen memory locations for the old and new ROMs are the same. These locations run from 32768, the home position, to 33767 for the lower right hand corner. This means that programs that PEEK or POKE screen memory should be OK on both PETs. The advantage of using these POKEs is that they are faster than the equivalent PRINT statements. The following program will print a screen border on both machines.

```
10 ?"(clear)"
20 FOR I =32768 TO 32768+39
30 POKE I,102 : NEXT
40 FOR I = 32808 TO 33728 STEP 40
50 POKE I,102 : POKE I+39,102
60 NEXT
70 FOR I = 33767-39 TO 33767
80 POKE I,102 : NEXT
90 GOTO 90
```

UPPER/LOWER CASE

There is one difference between the old and new ROMs that can cause problems and cannot be helped. When the CBM engineers changed the BASIC ROMs, they also changed the character generator ROM. This chip is the one in the far right hand corner of the main PC logic board. Its operation was significantly changed in the new PET versions. The table below shows the changes that were made.

POKE 59468,X		Old ROM	New ROM
X=12:	unshifted	upper case	lower case
	shifted	graphics	graphics
X=14:	unshifted	upper case	lower case
	shifted	lower case	upper case

As you can see when X=14 the upper and lower cases of the PET are reversed. What is even worse the ASCII codes for letters are also modified. In this mode on the old ROMs, ASCII 65 to 90 represents the upper case alphabet. In the new ROMs these numbers correspond to the lower case alphabet ! The conclusion is that programs that MUST have extensive lower case will look funny on the opposite PET.

There are other useful POKES and techniques that might be discussed in a follow up article. Let us know what you prefer to see. What kind of information do you need ?

Floppy Failures

Using unique disk IDs with the Commodore 2040 Dual Drive is a good idea for many reasons. In some cases, using unique IDs is more than just useful. When using two disks in the dual drive problems may occur if both IDs are the same. If the IDs are the same then only one Block Availability Map is written and data may be lost. In fact, both disks may end up with the same name and the same number of blocks free. A disk with only 100 blocks used could show only 200 or so free. The solution to this problem may be using the VALIDATE command. Better yet, use unique IDs.

Trouble-shooting your PET

by Doug Haluza

What do you do when you turn on your PET and its dead blank screen just stares back at you? There are some basic troubleshooting techniques you can use before you have to return it for service, if it's not still under warantee.

Naturally you shouldn't start digging into your PET before you've read this article at least once, and are sure that you understand the procedure. If you've never fixed anything before, or if you're not mechanically inclined you shouldn't start practicing on an $800 computer. Let someone else with experience do it, or return it to your dealer.

First be sure that it's plugged into a working outlet and has a good fuse. You can skip this step if you can hear the high pitched 'whine' the video monitor gives off, or if you can see the reddish glow from the CRT tube through the back of the monitor housing.

Turn the PET off and then quickly back on. If the monitor is working you should see a random pattern on the screen. If you don't, check the video connector after opening the PET; it's the one with 3 pairs of twisted wires comming out of it. It must make good contact. If this doesn't work give up. The problem is probably in the monitor, and there are always dangerous high voltages present there.

Unplug the PET before opening the case. Remove four screws from under the 'lip' of the PET (note: the newest PET's with molded cases only have two screws and a support stand in the front of the lower half of the case). CAREFULLY lift the white part of the case up (it's hinged in the rear). Watch for short wires, especially the casette cord on older 8K machines. Remove connectors if necessary noting how to reconnect them later. Take the support stand from the left half of the lower case and place it in one of the screw holes. This will hold the case open much like the hood on a foreign car.

If the CRT was on, but the PET still didn't respond, then check that all the IC 'chips' mounted in sockets are securely in place. Tap each one firmly, but don't break the PC board. Make sure you get all 23 in front, and the seven in back (new PETs only have nine, located in the rear). Also check that all four connectors are on securely. Check the power supply connector on old PET's for burns, especially if yor PET 'carshes' when you jar it. It's the connector with two brown, two red, and a black wire coming out of it--more on this later.

Close the lid, plug it in and try again. Your PET is probably cured. If not try again carefully rocking each chip in its socket. Don't try to lift it out because you'll probably bend the pins. Remember you're only trying to insure that they're making good contact with the socket.

As a last resort you can try swapping the scratch pad RAM's with the high RAM's. If they're bad the PET won't come up. Use the procedure described later in this article, but remember this is a long shot; the problem is probably in the ROM's.

If the power supply connector was burned it must be cleaned or replaced. To clean it remove the connector noting which way it was on. Now remove the small metal contact inside the burned section of the connector by pulling on that wire while pressing the small locking tab in back with a screwdriver. After it's out, clean the contact surface with fine emory paper and bend it out a little so it makes firmer contact, then put it back in the connector. Clean the lower half of the connector by lightly scraping each pin. Remove all filings to prevent shorts and lightly 'tin' each pin with solder if you can. Now put the top half of the connector on backwards. This way it'll make contact with the other side of the pins which should still be good.

If the problem persists lift the main logic board using the procedure described later. Check the solder joints for the power supply connector on the bottom of the PC board. The heat from the connector may have melted the solder and caused a bad connection. Resolder it if necessary.

If you were experimenting with the user port and are now having trouble with the casette decks, the user port, changing character sets, and/or the IEEE port, you may have blown the 6522 VIA chip. If you know how to remove and insert IC's try switching it with a 6522 from a working PET. Be careful to keep the chip away from static electricity sources like styrofoam. If this was the problem see your dealer about getting a new 6522 chip. It shouldn't cost more than $15.

If some of the characters on the screen flash or do other strange things the problem is probably in the screen RAM. New PETs have them soldered in place, so you'll have to return the board. On old PET's triple check the two IC's at board locations C3 and C4. Look down the edge of the PC board to find the letters and numbers just like you would on a map. If you're sure they're making good contact then switch them one at a time with the top set of program RAM's. Use the procedure below.

If your old PET shows less than 7167 bytes free, you probably have a bad RAM chip. Look up and triple check the two suspect RAM's in the table below for good contact. Turn it on again, and if it's still out swap them with the upper RAM's.

FREE MEMORY	2114	6550
7167 to 6144	JI-1	JI-8
6143 to 5120	JI-2	JI-7
5119 to 4096	JI-3	JI-6
4095 to 3072	JI-4	JI-5
3071 to 2048	JI-5	JI-4
2047 to 1024	JI-6	JI-3
1023 to 0	JI-7	JI-2
Scratch Pad	JI-8	JI-7
(Screen RAM)	C-34	C-34

This table shows the locations of the pair of suspect RAM's. Check the markings to see if they're type 6550 (24 pins) or 2114 (18 pins). RAM's marked TMS-4045 are the same as 2114's. Swap the suspect RAM's one at a time with the upper RAM's (7167 to 6144). Carefully lift the chip out of its socket by slowly prying it up on both sides. Be very careful not to bend the pins. When inserting the chips back in the sockets make sure the small key or dot faces the same way as the others for proper orientation. After you've swapped one, close the PET and turn it on. If the number of free bytes is larger, but not 7167, then that RAM is bad; mark it with red nail polish so you can replace it later. If the number is smaller, check the chips for proper alignment. If the number is the same, then swap the other one. If you were swapping the screen RAM the glitches will disappear and free memory will come down when you've found the bad one.

This procedure will give you the maximum amount of memory until you can replace the deffective RAM. If you get different numbers of free bytes when you turn on the machine, remove the bad RAM altogether; the intermittent chip may cause program errors.

You should be able to obtain either type of RAM chip from your dealer for under $15. The 2114 RAMs, however, are available from Radio Shack for $11. Ask for catalog no. 276-2504.

If your problem wasn't mentioned, or still isn't cured you may only have to return the main logic board for service. If the monitor works alright, the cassette deck is OK, and the black aluminum heat sinks get warm after the PET has been on for a while, then the problem is on the main logic board. You can remove it to return it to your dealer for repair or replacement by removing the 3 screws that hold it down and carefully lifting it off the 3 plastic locking tabs after removing all four connectors on the board along with any external connectors you may have added. Naturally you should check with your dealer before removing the board.

Time Passes Quickly

by Ralph Bressler

There are many reasons why you might want to measure time while using your PET. Sometimes you want to wait some time between two events such as the end of the instruction for a program and the behginning of its execution.In some programs you might want to know how long the user took to do something or limit the time for a certain problem. At times it is important to time the execution of statements in a program. Other steps in a program must be delayed to allow for human recognition and repsonse times.

The 6522 PIA device has several differnet clocks, some of which are assigned to reserved variables and others which may be accessed by using PEEK. Perhaps the easiest way to create a delay in aprogram is to use a simple time wasting FOR...NEXT loop. This is not exact and cannot be used to time events. An example:

```
100 PRINT"READ THIS QUICKLY SINCE IT WILL DISAPPEAR!"
200 FOR XX = 1 TO 2400 : NEXT
300 PRINT"(clr)I TIOLD YOU TO READ FAST!"
```

This program will print the first line, wait about 3 seconds, clear the screen and print the second line. By changing the upper limit of the loop you can control the delay. Each 400 iterations of the loop account for about .5 seconds.

On the PET the variable TI$ is resrved to measure time in hours, minutes and seconds. This value is stored as a string and represents time in the 24 hour system. The form is HHMMSS and to set the clock for 9:26 AM you might type the following directly on your screen.

```
TI$="092600"
```

If you set the clock when you turn the PET on, then you will be able to find the correct time by typing ?TI$ at any time. If you don't set the time then the string returned will represent the time since the PET has been turned on. To time the execution of a program you could:

```
10 TI$="000000"
 .
 .
300 ET$=TI$: PRINT ET$
```

This would give the elapsed time between statements 10 and 300 in the form above. Remember that this string may handled as any other. For example: LEFT$(TI$,2) would return the number of hours between statements or since the PET has been turned on.

To time shorter intervals the variable TI may be used. This clock measures time in 'jiffies' which are 1/60 th of a second. TI cannot be set direvctly but is affected by setting TI$. To time the above program in jiffies you might do the following:

```
10 TI$="000000"
  .
  .
300 ET=TI: PRINT ET
```

If PET prints 300 it means 300 jiffies or 5 seconds have elapsed between statements 10 and 300.

Both of these programs reset TI$ to zero and so interfere with the use of TI$ for keeping the time of day. To avoid doing this you can:

```
10 BT=TI
  .
  .
300 ET=TI-BT: PRINT ET
```

By doing this we record the time before and the time after the statements and subtract to get the elapsed time. This is a most common application of timing short time intervals. This method has a drawback related to the fact that if midnight strikes before the timing is done TI becomes 0 and the ET will be negative. To avoid this, add the following statement to your program:

```
310 IF ET<0 THEN ET=ET+24*60↑3
```

This adds 24 hours worth of jiffies to the negative number to get the true elapsed time in jiffies.

In common useage these are the only clocks needed. You might want to be aware that other clocks are available.

PEEK (59465)		counts in units of 256 microseconds
PEEK (59464)		counts in units of 1 microsecond
PEEK (512):	Old ROM	counts in units of 18 minutes,
PEEK (141):	New ROM	counts 0 to 80 in 24 hrs.
PEEK (513):	Old ROM	counts in units of 4 secs, counts
PEEK (142):	New ROM	0 to 255 in 18 minutes, increments
		every 256 jiffies
PEEK (514):	Old ROM	increments every jiffy, counts 1
PEEK (143):	New ROM	to 255 in 4 seconds

One interesting application is to use the WAIT command. For example for the Old PETs: WAIT 513,9 will wait a while and experimenting with the last number will increase or decrease the wait time. This command can be used with any of the locations above.

What Makes a Good Educational Computer Program

by JoAnn Comito

There are three key words in the title of this article: deucational, computer, and program. In order to answer the question posed in the title, the word 'good' must be associated with each of the three key words. In other words, a piece of software is a good educational computer program if 1)it is 'good' educationally, 2) it is 'good' to present the lesson on the computer, 3) it is a good program.

The lesson presented must be pedagogically sound. As educators you have been making this type of evaluation repeatedly; when looking at text books, filmstrips, preparing your own lessons, etc. A lesson presented on a computer should meet the same standards as a lesson presented via any other medium. The facts presented must be accurate. The concepts developed and/or reinforced must be clear and correct. Correct student responses should be rewarded with more spectacular results than wrong answers. To many programs use the reverse principle which actually encourages students to respond with incorrect answers. You are in the best position to evaluate the soundness of a lesson in your subject area according to your educational philosophy. If the computer program does not meet your standards, then it is not a good educational computer program.

The second key word in the question is 'computer'. A computer is not a textbook, nor is it a filmstrip projector or a tape recorder. It should not be used to imitate any of these items. It is a waste of time and effort to take a good lesson from a textbook and transform its o that the words appear on a TV screen instead of paper. The computer should be used to present lessons in ways that they cannot be presented in a textbook. The program sould take full advantage of the computer's capabilities: text and graphics displays, animation, sound, fast computation, and storage of large amounts of data. If an otherwise important concept is glossed over because the computations involved are too complex and numerous, then use the computer. If it is impractical to get hands-on experience, say running a nuclear reactor, use a computer simulation. If drill and practice lessons are boring to students, use the computer's graphics, sound and gaming capabilities to make these necessary drills more appealing. If the best way to present a topic is by elaborate animation, perhaps under student control, combined with immediate feedback is more appropriate, then use a computer. If the computer can enhance a lesson then use it. If you only use it to imitate other media already at your disposal, then you're wasting your time and probably money.

The last of the key wors is 'program'. In some ways good programming can be the most important of the three criteria, on the other hand it can be the easiest to correct if it is lacking. Once a lesson has been developed that will be presented on the computer, the equations worked out, the sequence of images determined, the type of animation decided upon, etc. iyt must be turned into a working program. Without this step there will be nothing for the student to see after turning on the computer. Of course, the lesson specifications can be turned over to a programmer for coding. Still, it is not enough to have a working program. A potentially excellent lesson can be ruined by poor programming practices. These programming pitfalls are in four areas; presentation of materials on the screen, processing student responses, inefficient programming and program maintenance. The first three problem areas are apparent when the program is running, the last when an attempt is made to modify the program. Again I would like to stress that while poor programming can seem to ruin a good lesson, the programming can be corrected by another programmer if need be, but no amount of programming can salvage a lesson that does not meet the first two criteria.

Some of the problems in the presentation of materials, or screen management are: not clearing the screen at the beginning of the program, allowing material to scroll off the top screen, forgetting to switch from lower case to graphics or vice versa, or forgetting to turn off the reverse field mode. While these problems are easy to correct, uncorrected they make a messy presentation.

Another programming problem concerns the way in which student responses are handled. The programmer must anticipate the types of answers the student will provide. The program must be able to hadle all these possible responses. Once traps have been set up for anticipated replies, another trap must be devised to handle the unexpected responses. If the program is expecting an integral input, be sure to check that the input is an integer. If you use the input as a divisor, be sure that it is not zero. Use an input routine that will not allow the student to fall out of the program if RETURN is pressed with no other input. After determining the type of inappropriate response the student has used, print a message explaining the response needed, don't leave the student guessing.

Inefficient programming may or may not be a problem, depending on how inefficient it is. There are two ways it can be inefficient: interms of speed of execution; or interms of memory used. If the program is short to begin with, then it does not matter if it wastes memory. On the other hand if the program threatens to exceed 7K, ways must be found to cut any waste of memory. Problems concerning execution speed are relatively rare, but if the programming is so inefficient that the student must wait for responses from the computer, then the inefficiencies must be removed. If there are so many calculations that long response times are unavoidable, then a message should be printed on the screen to that effect. It is much more comforting for the student to know that the computer is working and will return with a response, than to wonder if the system has crashed.

Problems of program maintenance will not be apparent to your students, but will be to you if you try to revise your programs. These problems deal with the structure of the program, use of REM statements, efficient use of subroutines, etc.

Good programming techniques can be learned. Development of good educational software depends upon your imagination and creativity. The microcomputer can be a very valuable classroom asset, use it to its full potential.

Writing That Good Educational Program

by JoAnn Comito

So you've come up with a terrific idea for an educational program, it is sound pedagogically and it will make good use of the computer's special powers. You also have an algorithm that will get the computer to carry out your idea. How do you turn your idea and algorithm into a good program ? There are several factors that will make a run of the mill program into a good program. The first factor

I will discuss is program organization.

Many beginning programmers are overwhelmed, or even frightened by the thought of having to write a whole program. There are PEEKs and POKEs, all kinds of variables, nested loops, cursor controls, and many other details to worry about. Once the program is written there may be problems debugging it or trying to modify it. There may be numerous GOTO statements transfering control to different parts of the program so you can no longer follow its logic. If you haven't worked on the program recently, you may find it very hard to remember what you did and why. The solution to these problems can be found in 'structured programming' and 'top-down design'.

When using the 'top-down' approach to programming, the large number of details that seem overwhelming are ignored until the very end. An example would be relocating from your present place of residence. You wouldn't worry about the color of the switchplates in the bedroom first. You would make the much broader, fundamental decisions first. You might decide on the country, the assuming you chose the USA, you would next select the state, an area and perhaps a specific school district. These first, top level decisions are relatively simple to make and many of them may be decided by external circumstances -- job location, for instance. The next level of decisions might involve choosing a price range for the house; a building and the number of rooms. It is not until after the simple, broad decisions have been made that you start worrying about decorating individual rooms. Even when you reach this 'room level' of decision making, you would not start with the color of the switchplates. You would start with a more basic decision, such as deciding between early american and french provincial. Design your program from the top down. Decide on the a major sections of the program, but don't worry about the details until later.

How do you decide what sections are needed ? Think of the program as a series of tasks that the computer must perform. Some of these tasks might include positioning the cursor; getting student input; printing corrective messages; displaying the gameboard; etc. How should these various sections be arranged ? Structured programming provides the answer here. The idea behind structured programming is to have a relatively short main program followed by a series of subroutines. Each subroutine performs one specific task. The main program acts as the conductor of a symphony orchestra; as a traffic controller; a general directing an invasion. Once the program has been orchestrated, the main program simply calls the subroutines as they are needed. The main program will look like an ordered list of tasks to be performed. There are not a lot of GOTO statements sending control all over the place. Control is switched to the subroutine, but as soon as the task is performed, control returns to the next statement in the main program.

Since each subroutine performs only one task, the subroutines are relatively independent of each other. Not all the rouitnes have to be developed for the program to begin running. At a very early stage of development the program might look like this:

```
80 REM MAIN PROGRAM
90 N=1
100 GOSUB 2000 @ PRINT  DIRECTIONS
110 GOSUB 1000 @ POSITION CURSOR
120 GOSUB 3000 @ GET INPUT
130 GOSUB 1000 @ POSITION CURSOR
140 ON N GOSUB 4000,5000 @ PRINT X/O IN BOX
```

```
150 GOSUB 6000 @ CHECK FOR WIN
160 IF N=1 THEN N=2: GOTO 110
170 N=1:GOTO110
200 END
210 :
1000 REM CURSOR POSITIONING
1010 PRINT"CURSOR POSITION"
1020 RETURN
2000 REM DIRECTIONS
2010 PRINT "HOW TO PLAY."
2020 RETURN
3000 REM INPUT ROUTINE
3010 PRINT "GET INPUT"
3020 RETURN
4000 REM PRINT X IN BOX
4010 PRINT "X"
4020 RETURN
5000 REM PRINT O IN BOX
5010 PRINT "O"
5020 RETURN
6000 REM CHECK FOR WIN
6010 PRINT "CHECK FOR WIN."
6020 IF WIN=1 THEN GOSUB7000 @ FIREWORKS .
6030 RETURN
7000 REM PRINT FIREWORKS
7010 PRINT "FIREWORKS"
7020 END
```

Even though none of the subroutines have been developed, the program can be run. You can check the flow of the program. You can make sure that the program actually gets to all of the subroutines at the right time. Now you can start working on each individual subroutine. As each subroutine is developed it will replace the PRINT statement that merely tells you that you reached the subroutine. You can write and debug each subroutine individually and place it into the structure of the program. Developing your program in this manner has several advantages. The details are left for the end and having each task of the program in a separate, labeled section simplifies debugging and independent subroutines can be used in different programs. There are several different methods of adding subroutines to main programs. The easiest method is using the Basic Programmer's Tool Kit. If you lack the Tool Kit see the article in this issue about merging programs.

Another important step in program development is the use of REM or remark statements to document your program within the program itself. These REMs are crutial if you do not intend to develop the program in one sitting. Without them you may not be able to tell what you were doing last time. Since these statements take up memory you may have to remove them as the program gets bigger. Still it is better to include them in the beginning.

Using the suggestions given here will make the development of those good educational programs much easier. The easier it is to program the more programs you can write.

A CREATIVE USE FOR COMPUTER MODELS

by James A. Fowler

For the last year I have used my PET to do basic research in developmental biology. Models of bilogical systems have been run on computers for many years. The usual purpose has been to test the effect of various assumptions and to get results to be compared with reality. Another and a rather difficult approach is to use computer models as a youngster uses some toys - to sharpen skills and to gain an insight into relationships embodied in the toy/model. For example, a toy crane will tip over if the boom is down and the load is heavy. Playing with a toy crane will give a 'feel' for the relationship: the more vertical the boom, the heavier the load can be without tipping over the crane. The mathematical analysis of this relationship is not difficult but the child need know nothing about it (nor, for that matter, need a crane operator know any trigonometry to operate his machine safely). Humans know a lot about nature. Only later does the theoretical analysis catch up to support this knowledge with theory. Not all knowledge comes this way, but it is a 'natural' way - a mode of investigation that is comfortable for our species and one that has been pretty successful.

One problem I work with is the relationship between cells of an organism which result in the development of patterns. We know cells have identical genetic programs guiding their behavior by directing their interactions with their environment (which, of course, consists mostly of other cells). But in spite of their identical instructions, they behave as if each cell had unique personal instructions like players in a marching band - the outcome is a predictable pattern of high precision that appears in spite of perturbations from outside the organism. This is impossible to explain in light of present knowledge even though we do know at least some of the processes going on inside cells.

The model for this situation is a 'cellular automaton'. This is an array (I stick to two-dimensional sheets) of individual 'machines', each controlled by some sort of programmed interaction with the environment (which consists partly or entirely of other such machines). So far the results have been encouraging. The main difficulty is presenting the user with important information in such a way that one can develop the rules resulting in predictable patterns. The rules are embodied in a 'transition matrix' (really just a fancy table) common to all cells. The matrix allows the cells to find out what to do next under all sets of possible environmental parameters and all possible internal states. Our program displays the 'state' of each cell as an integer (0-9) on the screen. We can transform integers to symbols upon command so that the eye can easily see patterns develop. A second display shows the state of a 'clock' within each cell - biological rhythms are probably avery important element in the devlopmental sequence of animals and plants. The transition matrix is also displayed on command (at least one page at a time, because the whole matrix has too many elements to fit on the screen comfortably). Any of these displays can be modified by the users as easily as you can modify text on the PET's screen. The modifications then become part of the model from that moment on. The goal of the user is to make a transition matrix that meets certain requirements of pattern, stability, and response to operative interference. In other words, the model mimics nature.

The work is still in an early phase. Several undergraduates at Stony Brook have done independent work on one or more of these programs and have developed transition matrices for a number of interesting and significant patterns. Interestingly enough, graduate students have not shown as much flexibility and interest in this work - and, of course, faculty are even more mystified by the whole idea. We started with several programs in FORTRAN and used a large minicomputer (MODCOMP IV) to run them.

Our real progress began when we transferred the work to a PET 2001-8. The rich graphic repetoire and screen addressable feature made the PET the best personal computer for the job. We have gone through many versions of the original programs. More and more of the routines are written in assembly language. Our latest version is entirely in assembly. It is a joy to run (except for a bit of nuisance in the FILE, LOAD, SAVE routines which Commodore never meant for us to mess with). The speed of the routines is so great that we have had to put in WAIT sequences to slow things down. We are going to play around a bit with speed verus user responses this coming year. There is probably an optimum range. After all, the kid playing with the toy crane wouldn't learn very much if the crane either exploded or the load flew to the top of the boom at the instant the kid touched the controls !

I have learned many things from writing these programs: For example, it is worth spending a lot of energy and ingenuity on ways the user can erase something just given to the program. You must be able to 'back up' so to speak even though it is rarely needed. Remember how you felt when you took the wrong turn off the parkway ? You could not back up and corect your error. You were penalized by the system ! you had to continue down the wrong road until you could figure out some way to get back on the track. Nothing can turn off the enthusiastic user faster than not being able to escape from some run that has obviously gone wrong. When you write assembly routines you have to provide all the polite responses the PET makes when you are working in BASIC such as allowing you to delete or change the response to an INPUT. We have also found that a program to be used repeatedly can (and should) have fewer and shorter prompts - but they should be consistent in requiring the same kind of response in each case. If you call for a number in one case and a letter in another, make the prompts very different in appearance by making one in capitals and the other in #? or something like that. Users learn to work very fast but make mistakes if their responses are not properly cued.

If you are interested in more details of this work, with reference to biological works, there is a summary in the PERSONAL COMPUTING PROCEEDINGS of the AFIPS NATIONAL COMPUTER CONFERENCE, 1979, pages 187-191. I would be happy to talk to anyone interested in trying out computer-run models in this mode.

Pretty Printer Listings

To list a program to a printer connected to the PET's IEEE port try:

```
OPEN4,4:CMD4,"Title":LIST
```

This will provide a title for your listing. On the PET/CBM printer you can enhance the title by using:

```
OPEN4,4:CMD4,CHR$(1),"Title":LIST
```

PET Files

by Ralph Bressler

After using the PET for a while you begin to realize that to exploit its full potential you need more than the standard BASIC commands. PEEKs and POKEs become second nature and even a few short machine language routines may creep into your programs. The PET file commands are indespensible when using almost any of the available peripheral devices. These commands are necessary for listing prigrams on the printer. The PET printers also make extensive use of these commands for formatting data sent to them. File commands must be used when employing the cassette tape deck for data storage and the disk drive for program recording or data storage.

This article assumes that not everyone has $1295 for a disk but may want to store data on tape. I, therefore, have concentrated on the use of the file commands with the tape deck. Since printers and disks are beginning to find their way into the hands of more people an article dealing with these devices and their file commands would be appreciated by many users.

The PET BASIC file commands are listed below. In these commands LF stands for logical file number, DV means device number, IO is the input or output option, FN is the file name, and VL is the variable list.

OPEN	Initializes a file for use by the PET
CLOSE	Tells PET to remove a file from use
PRINT#	Writes data to a file
INPUT#	Reads data from a file
GET#	Gets one character at a time from a file
CMD	Sends BASIC's output to a file
ST	Records the status of I/O operations

```
OPEN LF, DV, IO, FN
```

This command tells the PET to send information to a particular device or retrieve it from that device. Only the LF must be given but it is best to specify all the parameters. LF can be between 1 and 255 but only 10 files can be open at once. This is many more than you will ever need to open at one time. DV tells the PET which device to address as follows:

0 = PET keyboard
1 = first cassette drive
2 = second cassette drive
3 = PET screen
4 = PET printer
8 = PET disk.

Actually 4 to 15 address the IEEE port

If not specified DV defaults to 1, or the first tape unit. For the I/O option IO may have the following values:

0 = read file only.
1 = write file only.
2 = write file with an End Of Tape marker.

The FN is used to name data files when they are recorded on tape. It is not necessary to use a file name but it helps prevent reading the wrong data file. If FN is specified as 'Class Data' the PET will search a tape until it finds the file by that name. The LF, DV, IO can be computed expressions and FN can be a string variable. OPEN X*Y, 1, A%, WE$ is legal. some common examples follow:

```
OPEN 1,1,1  OPENs file #1 to write to tape deck #1
OPEN 1,1,0  OPENs file #1 to read form tape deck #1
OPEN 4,4    OPENs file #4 to output to the printer
```

Remember not to forget commas. Also, if you want to specify a file name you must include all other parameters.

CLOSE LF

This command removes the file from use and outputs any characters stored in the buffer to the tape and writes and end of file marker if the tape was being used. To write to a file and then read from it later you must close it first. Failure to CLOSE a file before OPENing again can be a fatal error.

PRINT# LF, VL

This statement outputs the variable list to the indicated file. The file must be OPEN and IO must = 1 for writing. Be sure to spell out PRINT# because ?# is not legal. This command will write into the file exactly what PRINT would put on the screen if you handle it correctly. It is good practice to follow a PRINT# command with a PRINT just to check to see what data is being sent. Examine the following examples:

```
OPEN 4,4: PRINT#4, X$     Will print on the printer the
                          string represented by X$
OPEN 1,1,1: PRINT#1,A     Will write the number represented
                          by A onto tape deck #1
```

INPUT# LF, VL

INPUT# reads data from a file and assigns the data to the variable indicated. The file must be OPEN to read (IO=0). There are three problems which may interfere with successfully reading data from a tape. What is read using INPUT# must be EXACTLY what was written using the PRINT# command. If there is a difference an error will occur. Do not INPUT# a string of characters over 79 in length since this will close the file and not allow any more operations. Finally, you may use INPUT# to read more than one variable on a line but do not use PRINT# to write more than one.

```
INPUT#2, N, N$, FG     This will read the variables
                       indicated with no problems.
PRINT#2, N, N$, FG     This will cause the above INPUT#
                       to malfunction and DATA will
                       not be read correctly.
PRINT#2, N             This is the correct way to write
PRINT#2, N$            the three variables onto tape so
PRINT#2, FG            that INPUT# will read them.
```

GET# LF, VL

Just like the GET command this will grab one character from a file at a time. The best way to use this is to only GET# a string variable since a number can be read as a string but not vice versa. This command works well for reading DATA from any file especially when you aren't quite sure waht the data is. GET# will not stop automatically at the end of a file. To solve this problem always write a character at the end of your data files and then check for that character.

CMD LF

This command allows the PET to communicate directly with a file. When BASIC does something it sends what it has done to the operating system. The operating system usually responds by printing information on the screen. CMD redirects BASIC's output to a file. For example:

```
    OPEN 4,4 : CMD 4 : LIST
```

will cause the listing to be output to the printer and not to the screen. After you are through create a SYNTAX ERROR to get things back to normal.

ST - Status word

This variable is reserved to tell the status of any I/O operation after it has been performed. For example, after loading a program from tape ST should equal 0 if there were no errors in reading. The values ST may take during tape I/O are:

4	short block	tried to read program as data
8	long block	tried to read program as data
16	unrecoverable read error	PET unable to read your tape,
32	checksum error	clean tape heads
64	end of file	useful to determine if you
128	end of tape	are at end of file

To check the end of a tape you could use IF (ST) AND 128 THEN....
The following program illistrates the use of some of these commands. It also corrects for two problems which plagued old PETS but were corrected in the new.

```
10 X$="DATA"
100 POKE 243,122 : POKE 244,2
105 OPEN 1,1,1
110 FOR N = 1 TO 500
120 PRINT#1, N : GOSUB 500
130 PRINT#1, X$ : GOSUB 500
140 NEXTN : CLOSE1 : PRINT"REWIND & PRESS ANY KEY"
150 GETX1$:IF X1$="" THEN 150
160 OPEN 1,1,0
170 FOR N = 1 TO 500
180 INPUT#1, X, Z$
190 NEXTXN : CLOSE1 : PRINT"END OF DEMO" : END
500 REM***BUFFER CHECKER***
510 IF Z9 <= PEEK(625) THEN 530
520 POKE 59411,53 : FOR Z9 = 1 TO 140 : NEXT : POKE 59411,61
530 Z9=PEEK(625):RETURN
```

Line 100 makes sure that the data file header is written correctly. The subroutine at 500 to 530 is also important. When using PRINT# PET does not immediately write data on tape but stores it in a buffer. This buffer can contain 191 characters and when it is full the PET dumps it onto the tape. Between each 'dump' of the data is an interblock gap, a physical space on the tape. This gap is important when reading the data back. Sometimes the gap is too small and the data written cannot be read back correctly. The subroutine included increases the gap between blocks. For the second cassette the following changes should be made:

```
100 POKE 243,58 : POKE 244,3
510 IF Z9<=PEEK(626) THEN 530
530 Z9=PEEK(626):RETURN
```

The program below is from Commodore and will show the data from any data file in 80 character hunks.

--SHOW TAPE--

```
100 PRINT"❐   --SHOW TAPE---▨
110 PRINT"PUT YOUR DATA TAPE IN
120 PRINT"CASSETTE #1 AND REWIND IT.
130 GOSUB 430
140 PRINT"▨THE TAPE WILL BE READ AND SHOWN TO YOU
150 PRINT"IN 80 CHARACTERS HUNKS. WHEN YOU WANT
160 PRINT"TO STOP PRESS ANY KEY. IHE PROGRAM
170 PRINT"ASK IF YOU WANT MORE DATA TO BE SHOWN.
180 GOSUB 430
190 OPEN 1
200 PRINT"❐":H=0
210 H=H+1:PRINT"HUNK #"H
220 FOR J= 1 TO 80
230 GET#1,B$
240 IF ST > 0 THEN 350
250 IF ASC(B$)=13 THEN PRINT"  ";:GOTO270
260 PRINTB$;
270 NEXT J
280 PRINT
290 GETA$
300 IFA$=""THEN320
310 PRINT"MORE?";
320 GETA$:IFA$=""THEN320
330 IFA$="Y"THENPRINT:GOTO210
340 END
350 PRINT:PRINT"STATUS WORD IS:"ST
360 IF (ST) AND 4 THEN PRINT"SHORT BLOCK
370 IF (ST) AND 8 THEN PRINT"LONG BLOCK
380 IF (ST) AND 16 THEN PRINT"READ ERROR
390 IF (ST) AND 32 THEN PRINT"CHECKSUM ERROR
400 IF (ST) AND 64 THEN PRINT"END OF FILE
410 IF (ST) AND 128 THEN PRINT"END OF TAPE
420 END
430 PRINT:PRINT"PRESS ANY KEY
440 GETA$:IFA$=""THEN440
450 PRINT:RETURN
```

Merging Programs

by JoAnn Comito
Ralph Bressler

The ability to merge programs is of great importance to the serious software developer. Being able to combine to programs or add pre-programmed subroutines saves time and frees the programmer for more creative work. Before discussing some of the ways to combine programs, it is important to define two terms. Merging can indicate two separate and different functions. Appending means adding code to the end of a program only. Appending will not place lines 50-100 in the middle of the program. Weaving will perform this function. as well as appending and is, therefore, the more powerful tool.

The simplest way to weave requires no extra hardware and no special preparation of your prepared subroutines. Program your subroutines so that they fit on the top 18 lines of the screen when listed. Now follow the procedure outlined below after recording a copy of your routine on tape:

1) Home the cursor and LIST the program; scroll the LISTing to the very top of the screen

2) Place the cursor on top of the READY and type LOAD. Press PLAY when asked and then hit RETURN.

3) When the READY is given home the cursor and hit RETURN for each line of code on the screen.

Again, this method requires no special harware or software and will weave your subroutine into the main program. The big disadvantage is that the routines must be very short.

Another way to weave programs is to use the Librarian program by D. J. David in the April 1980 issue of Microcomputing (p. 172). This amazing 25 line program allows you to program your routines and then record them using the special method in the program. This saved routine is not compatible with the normal PET save but is read by the Librarian program. This program resides at the beginning of any program and performs several useful functions. It has automatic line numbering as well as the weaving feature. Librarian will also save a part of a program bewteen certain line numbers. For the program and a complete description see the article mentioned.

Another way to merge is to use the MERGE program in the February issue of CURSOR magazine. This program will perform any weave or append and works with normally constructed and SAVEd programs. This program does require the Commodore 2040 Dual Floppy Disk Drive. If you already have the drive the program is great but $1300 would be a lot to pay for this function alone.

There are two remaining ways I know of to append subroutines to existing main programs. In reality, appending is quite adequate since you are only adding subroutines in many cases. Subroutines can and should go at the end of the program anyway. (See the article '...Good Educational Program' in this issue.) The Basic Programmer's Tool Kit is the easiest way to append that I know of. This piece of firmware from Nestar Corporation costs bewteen $50 and $80. The 'chip' adds 10 commands to your PET and by typing APPEND you can simply add subroutines directly to the end of any program. This requires no special preparation of the routines which can also be

loaded as normal programs.
The last method of appending programs or subroutines again requires no special hardware but does require that the routines be specially prepared. Once the tapes have been prepared the routines can be added to any program. The special preparation, therefore, is not a great disadvantage. Follow the directions below exactly as written. This method is contributed by Larry Tesler and Jim Butterfield.

Step 1: Preparing the Subroutine

a) Enter the routine from the keyboard or LOAD it from tape.

b) Put a blank tape in the cassette recorder.

c) Type 'OPEN1,1,1:CMD1:LIST' and hit RETURN.

d) Now place RECORD and PLAY.

e) When the cursor appears type '?"POKE611,0":PRINT#1:CLOSE1' and hit RETURN.

Step 2: Merging Subroutines

a) Enter or LOAD the program to which the routines are to be merged.

b) Place the tape with the prepared subroutine from Step 1 in the recorder.

c) Type 'OPEN 1' and hit RETURN.

d) Press PLAY.

e) When the tape stops type [clr][4 down].

f) Type 'POKE611,1:POKE525,1:POKE527,13:?"[home]"' but RETURN.

g) Type '[home][6 down].

h) Type 'POKE611,1:POKE525,1:POKE527,13:?"[home]"' and hit RETURN.

The tape should start and continue until a ?SYNTAX ERROR or ?OUT OF DATA message appears between the two lines you entered. If a message does not appear after a reasonable time press RUN/STOP. The merge is now complete.

i) Type 'CLOSE 1' and hit RETURN.

You can now LIST the program and see that the append has been done.
Using one or several of these techniques you can develop your own subroutines and add them to your programs as needed. The Program Exchange has many of these subroutines available under the titles of Useful Rot 1 and Useful Rot 2. Each package contains 5 to 7 routines including auto line numbering and deleting and repeating keys.

An 80 by 50 Plotting Routine

by Doug Haluza

Although the PET does not have a built in high resolution mode, reasonable plots can be obtained by using the quarter square blocks. Using them quadruples the PET's resolution giving an 80 by 50 array of points for graphs, charts, games, etc.

Using this method is made easy with the routine called by the demonstration program shown in Listing 1. After the array T%(15) has been properly initialized, calling the subroutine in line 50 with the X and Y coordinates in the variables X and Y will either set or reset that point, depending on the value of Z. If Z is positive, the point will be turned on (set); if Z is negative the point will be turned off (reset). Nothing will happen if Z is zero.

The routine works by assigning a value to each quarter character block as shown in Figure 1. Adding the values of the blocks to be turned on in any character gives the character code. The array T% contains the POKE values for the correct characters for each character code.

Setting or resetting any particular block is done by adding or subtracting the value of the block from the current character code. Care must be taken to assure that the block to be set (or reset) isn't already set (or reset).

The routine in line 50 starts by finding the location of the character containing the block to be plotted and saving it in L. The loop looks up the character code and stores it in K.

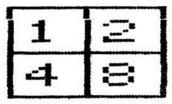
In line 51 I is set to the value of the block to be plotted. '((KANDI)>0)=(Z<0)' checks to see that the block isn't already plotted; if not it is plotted by adding or subtracting the block value from the character code and POKEing the correct value back into location L.

When using the subroutine in another program remember to set X,Y and Z; include line 5; and be sure to keep the DATA in line 10 separate from any other DATA the program needs.

Listing-1

```
1 REM 80 BY 50 PLOTTING ROUTINE
2 REM BY DOUG HALUZA
3 PRINT"❐";
5 DIMT%(15):FORI=0TO15:READT%(I)
6 NEXT:Z=1:REM **SETS UP DATA TABLE**
10 DATA32,126,124,226,123,97,255,236
11 DATA 108,127,225,251,98,252,254,160
12 REM ***DEMONSTRATION***
15 X=X+1:Y=-20*SIN(X*.075)+20.5
16 GOSUB50:GOTO15
18 REM ***PLOTTING SUBROUTINE***
50 L=32768+INT(X/2)+40*INT(Y/2):K=0:I=PEEK(L):FORJ=0TO15:IFI=T%(J)THENK=J
51 NEXT:I=2↑((XAND1)+(YAND1)*2):IF((KANDI)>0)=(Z<0)THENPOKEL,T%(K+I*SGN(Z))
52 RETURN
```

Figure 1



Peeking at BASIC

by Doug Haluza

Anyone who has tried PEEKing at BASIC on an old PET knows that you always get a 0 no matter what location you 'look' at, whether that location contains a 0 or not. Commodore and Microsoft did this to try to protect their BASIC. You may be able to understand why this happens by looking at the disassembly listing from the PET below:

```
D6E6:    20 D0 D6    JSR D6D0      Evaluates a formula an if it's
                                   0 to 65535 puts it in 8,9
D6E9:    A0 00       LDY #0        Zero Y register
D6EB:    C9 00       CMP ##C0      Checks if MSB is between $C0 and
D6EB:    90 04       BCC D6F3      $E1 (eg. trying to PEEK at BASIC)
D6EF:    C9 E1       CMP ##E1      and if so puts 0 in Y. BASIC is
D6F1:    90 03       BCC D6F6      locations $C000 to $E100
D6F3:    B1 08       LDA (08),Y    Does the PEEK
D6F5:    A8          TAY           Put it in Y
D6F6:    4C 87 D2    JMP D287      Load the floating point
                                   accumulator with Y
```

You see in $D6EB that a check is made to see if you're trying to look at BASIC and doesn't do the PEEK if you are. This can be easily defeated by the routine below:

```
03F8:   20 D0 D6    JSR $D6D0     Evaluate expression
03FB:   A0 00       LDY #0        Zero Y
03FD:   4C F3 D6    JMP $D6F3     And jump in after check
```

This routine is put into the very top of the second cassette buffer and can be loaded using the monitor and typing over the line so it looks like this:

```
.M 03F8 03FD
.:03F8  20 D0 D6 A0 00 4C F3 D6
```

or by using the following BASIC routine and USR instead of PEEK.

```
10 DATA 32,208,214,160,0,76,243,214
20 FOR I = 1016 TO 1023: READ N: POKE I,N: NEXT
30 POKE 1,248: POKE 2,3
```

PET's Round off Problems

by Raplh Bressler

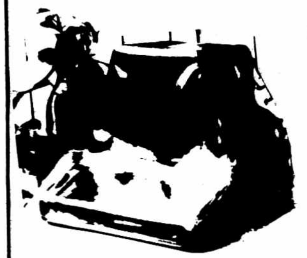
The PET has some serious problems with seemingly simple math problems. The simplest illustration of this problem is to type: '? 7*7,7↑2'. The PET will print '49   49.0000001'! Because the PET uses natural logs to do exponentiation (and square roots) an error of a few bits was introduced. In this case the error was large enough to show, but in some cases it isn't.

I was writing a program that was supposed to print Pythagorean triples such as 3 4 5. These are right triangles whose sides have integer lenghts. The program was supposed to print the value of the sides of the triangle and then print a star if it was a triple. To check the value of the hypotenuse to see if it was an integer I used the statment:

```
100 IF INT(HYPO)=HYPO THEN PRINT "*"
```

Imagine my surprise when 3 4 5 appeared without a star! What was more amazing was that the PET printed 5 for the value of HYPO but did not recognize it as 5. I've found the following statment works better:

```
100 IF STR$(INT(HYPO))=STR$(HYPO) THEN PRINT "*"
```



ITS COVERS . . .

A BEAUTIFUL COMPLEMENT TO ANY DECOR!

- Attractive, Durable, Leather-Grained Vinyl
- Double Stitching for Extra Strength
- Corded Seams Assure Perfect Fit
- Colors – Gold, Chestnut Brown, Olive Green or Black

PET/CBM – $18.95
Apple II – $12.95
Apple Disk – $8.95

OTHER COMPUTER COVERS AVAILABLE. WRITE FOR DETAILS

NEW-CURSOR FOR PET 2001 OLD ROM
INSTANT PUSHBUTTON CURSOR RETRIEVAL

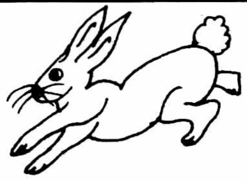
$7.95

- Reviewed in Issue #1 of *COMPUTE*
- Restart without touching power
- Machine language programs in second cassette buffer not lost.
- Simple, illustrated instructions
- Installed in minutes – no soldering
- Improved mounting – no sticky tape

UNCRASHER for PET/CBM 2001 - New ROM — $14.9F

INTERNATIONAL TECHNICAL SYSTEMS, INC.
Box 264-a Woodbridge, Virginia 22194
Phone Orders (804) 262-9709
VISA master charge
SHIPPING AND HANDLING ADD $1.00

PET RABBIT



Load, Save, Verify, Execute
8 K in 38 seconds versus
PETs 2 Minute 45 seconds,
plus more!

High-speed Cassette Routines work with 8K, 16K, or 32K **new ROM PETs** which have the new Commodore cassette deck (like the external version which sells for **$95.00**). Note: If you have a new ROM PET with the old style lift-top deck, everything but the high-speed cassette routines will work.

— **Auto repeat** of any key held down, toggle character set.
— **RAM Memory Test**, convert #'s to hex and decimal.

12 Rabbit Commands

Note: Rabbit is 2K of machine code at **$1800** for 8K PETS, **$3000** or **$3800** for 16K PETS, or **$7000** or **$7800** for 32K PETS. (Specify one of the 5 versions.)

Cassette and Manual — $29.95 (Add **$5.00** for foreign)

Eastern House Software

3239 Linda Dr. **Winston-Salem, N. C. 27106**

Finally, MAE – A PET DISK-Based MACRO ASSEMBLER/TEXT EDITOR

Works with 32K PET

— Works with 2040 Disk, and can drive 2022/2023 Printer, and/or RS232/20 ma Device thru User Port.
— 100% Disk Based, 100% Machine Language.
— Macros, Conditional Assembly, and a new feature we developed called Interactive Assembly.
— Coexists with Basic, Auto character repeat, Sorted Symbol Table.
— 27 Commands, 26 Pseudo Ops, 5 Conditional Ops, 38 Error Codes.
— Creates relocatable object code on disk.
— Assemble from Memory or Disk.
— String search, search and replace, and inter-line edit.
— Auto line #-ing, move, copy, delete, renumber.
— Labels up to 31 characters — user specifies length.
— Includes extention to PET monitor (disassemble, trace, etc.), Library of PET ROM locations, Relocating Loader, plus more.

Manual, Diskette, U. S. postage — $169.95
(Requires completion of License Agreement — Write for details)

Eastern House Software

3239 Linda Drive **Winston-Salem, N. C. 27106**

PET/CBM UNCRASHER™

WHAT IS IT? — UNCRASHER™ is a two button device that allows PET/CBM users to regain control of a cursor that's been lost due to programming errors. BASIC programs may be recovered. Machine language programs in the second cassette buffer are not disturbed either.

WHICH PETs/CBMs? — UNCRASHER™ is for all PET/CBM computers that use the "NEW" Version 2 ROMs. (Older model PETs should use the ITS NEW-CURSOR™.)

DOES IT WORK? — You bet!!! See the detailed review of the types of crashes and the concept of recovery in the first issue of *Compute*.

INSTALLATION — Simple, completely illustrated instructions using only a Phillips screwdriver ensure installation in minutes. No soldering or modificaitons to the computer.

OPERATION — Just follow the simple steps—push the buttons and reset the stack pointer—and PRESTO . . . recovery!

And all this happens without powering the PET/CBM down and up.

WHY UNCRASHER™? — No first class computer such as the PET/CBM should be without this capability. Whether your fancy be programming, business, education, or hobby, hobby, UNCRASHER™ saves you time by uncrashing your slip-ups.

AVAILABILITY — Now in better computer stores, or order direct from ITS, made by the people who brought you NEW-CURSOR™.

INTERNATIONAL TECHNICAL SYSTEMS INC.
P.O. BOX 264 -a WOODBRIDGE, VIRGINIA 22194
CUSTOMER SERVICES –
RICHMOND, VIRGINIA (804) 262-9709
VISA master charge

COMPUTER PROGRAMMING:
BASIC for Microcomputers

Bridge the Classroom Literacy Gap.

A series of five full-color filmstrips and five cassettes prepared by Arnold and Seth Friedman. This is a systematic approach to the essentials of the BASIC language.This series can be used at any grade level to teach the universal concepts which apply to the PET, TRS-80, Apple or any microcomputer.

PART I	Getting Started
PART II	Mathematical Operations
PART III	Loops and Subroutines
PART IV	Original Programming
PART V	Techniques and Flow Charts

Available for $84.00 for the set. Also available for preview.

EDUCATIONAL ACTIVITIES
Freeport, NY 11520

PET™ 2001 **STUDENT WORKBOOKS**

For Classes Beginning to Program

COMPLETE TEACHING UNITS
- Step-By-Step Instruction (Teachers Can Be Beginners, Too!)
- Daily Lesson Plans, Classwork & Homework Worksheets, Quizzes
- Hands-On and Hands-Off Exercises
- Group and Individual Assignments

FEED ME, I'M YOUR PET™ (Beginner-1)
LOOKING GOOD WITH YOUR PET (Intermediate-1)
TEACHER'S PET (Plans, Quizzes, Answer Key)
Workbooks are $4.35. Quantity discounts available.
TEACHER'S PET is $4.00 or **FREE** with workbook orders of 25 or more.

EDUCATIONAL SOFTWARE
- High Quality Teaching Skills and Programming Skills
- Graphics Used to Enhance Learning

REMEDIAL MATH PACKAGE (84 Concepts)
MATH IN SCIENCE SERIES

Signed Numbers, Physics, Primary Primer, Elementary Grade topics and More. We Will Send a Price List.

COW BAY COMPUTING
Box 515 • Manhasset, NY 11030
F.O.B. Manhasset, NY

B.C. COMMUNICATIONS

The Family Computer Shop

- Software Swapping
- Factory Authorized Service Center (Same Day Service)
- Commodore Word Pro III
- Word Processing Service
- Computer Products leasing with buy in percentage
- Factory trained personnel
- Centronics Printers at very special prices
- Consummable supplies at discounts
- Reference manuals and magazines available
- Atari and APPLE III ''hands-on'' demos
- Commodore's new 80 character Computer
- NEC & Qume Electronic typewriters
- Special interfaces

207 DEPOT ROAD, P.O. BOX 228
HUNTINGTON STATION, NEW YORK 11746
TELEPHONE: AREA CODE 516□549-8833 or 692-2735

Riley Enterprises announces....

TOP Quality Vinyl Cassette Holders
12 Capacity in Black or Brown $2.80
16 Capacity in Blk,Org or Lime 3.50

Add $2 for shipping orders under $20

Other types and colors are special order.

75 Pearson Street
Portsmouth, NH 03801 603-436-6564

Programs !!

•Business Research	$50
•Home & Small Business	$15-40
•Games & Simulations	$15 ea.
•Education (HS) Pack	$15

Send for complete catalog!

•Shopper	•Home Addresser	HARRY H. BRILEY
•Inventory	•Deluxe Addresser	Catalog Info.
•Museum!	•Dinner's On!	P.O. Box 2913
•Mansion!	•Fur Trapper	Livermore, CA. 94550
•Pentagon!	•High Seas	(415) 455-9139

AMERICAN PERIPHERALS

First in Education

Have You Seen?:

- MTU's new Visible Memory
- 3M's Datronics card reader with our special test marking software
- Commodore's new 80 column CBM with BASIC 4.0 and WordPro 4
- Centronics' new 737 high quality printer
- Xerox's Diablo printers for letter quality output

Did You Know?:

Atari now has a special 3 for 2 deal for schools

Send for a complete list of our full line of educational and business software.

3 Bangor Street
Lindenhurst, NY 11757
(516) 226-5849

COMPUTER LAND of Nassau

"The Full Service Computer Store"

Come **SEE** what you've only **READ** about.

Hardware...Apple...PET...Atari...Texas Instruments...Cromemco...Dynabyte...computers for personal and professional needs...Diablo...Centronics...Spinwriter... printers and disk drives to complete a system.

Software...programs for **ALL** the machines we carry...programs for recreation, education and business.

Books and Magazines...Osborne...Blacksburg Group...Hayden...Sybex...Scelbi...or pick up a copy of your favorite magazine.

Computer land
79 Westbury Avenue
Carle Place, NY 11514

CENTERBROOK SOFTWARE DESIGNS

"Learning is Fun"

Remedial Math and Reading

Science, Utilities, and Games

We have over 70 programs in the topics listed above. All programs work in 8k and there are versions for all PETs. Prices for most programs are in the $10 to $20 range and special packages are available. New programs are produced every month and custom programming is available.

For a catalog and more information write or call:

Centerbrook Software Designs
98 Emily Drive
Centereach, NY 11720
516-585-2402

American Peripherals
3 Bangor Street
Lindenhurst, NY 11757
516-226-5849

WEST RIVER ELECTRONICS

Can Your PET accurately represent complex mathematical symbols such as integration or exponentiation?

Can you write equations which include Greek symbols and closed intervals?

Would you be able to have your PET display French, Spanish, German or other European languages as they are written?

If you answered no to any of these questions then you need our PET Alternate Character Set ROMs. Our replacement character generator chips do all of the above. In the UPPER/lower case mode the shifted upper row of characters and the numbers are redefined. There are two ROMs, one for the foreign languages and one for math symbols.

Math Character ROM	$75.00
Foreign Language ROM	75.00

PO Box 605
Stony Brook, NY 11790

QUALITY SOFTWARE FOR THE PET*

8K or more, model 2001-8

*PET is a product of Commodore Business Machines, Inc.

✦ **MEM-EXPLORER™** $7.95

✦ **SWEE-PING™** $5.95

✦ **CHEQUE-CHECK™** $7.95

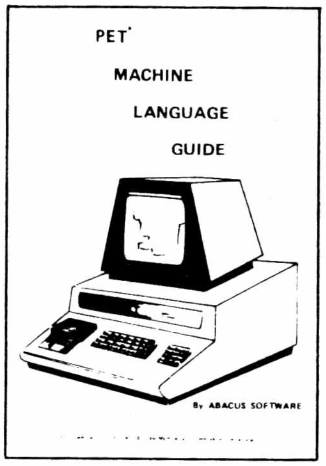
✦ **METRIC-CALC™** Turn your PET into a powerful stack-oriented (RPN) calculator with many extras! Log, trig, exponential, and many other useful scientific functions in addition to Metric-English unit conversions. Switch between two "keyboards" at the touch of a key. See the stack while you operate. Look at all 20 addressable memories at one time. More functions than calculators costing many times as much. Unlike other converters, this one lets you use results in other calculations! $7.95

✦ **BILLBOARD™** Turn your store window or counter into an attention-getting advertising display! With BILLBOARD, you simply type in the message of your choice (up to 254 characters), then see it march across the screen in giant, one-inch high letters ... even pause, or flash off and on if you like! A real crowd-stopper, and it costs less than a single ad. Order yours today! $49.95

MICRO SOFTWARE SYSTEMS
P.O. Box 1442, Woodbridge, VA 22193

Send check or money order. Allow two weeks for check to clear. VA residents add 4% tax. Dealer inquiries invited.

PET' MACHINE LANGUAGE GUIDE


PET'
MACHINE
LANGUAGE
GUIDE
By ABACUS SOFTWARE

Contents include sections on:
- Input and output routines.
- Fixed point, floating point, and Ascii number conversion.
- Clocks and timers.
- Built-in arithmetic functions.
- Programming hints and suggestions.
- Many sample programs.

If you are interested in or are already into machine language programming on the PET, then this invaluable guide is for you. More than 30 of the PET's built-in routines are fully detailed so that the reader can immediately put them to good use.
Available for $6.95 + .75 postage. Michigan residents please include 4% state sales tax. VISA and Mastercharge cards accepted - give card number and expiration date. Quantity discounts are available.

ABACUS SOFTWARE
P. O. Box 7211
Grand Rapids, Michigan 49510

• PET • PET • PET • PET • PET • PET • PET • PET • PET • PET • PET • PET • PET • PET

PET PRODUCTS

Programs — Workbooks
for Floppy Disk — for Cassette

Dealer Inquiries Invited

Put your PET to work!

T I S

PROGRAMS		WORKBOOKS		
SW-1*	**MAILB** mailing list system	WB-1	**Getting Started with Your PET**	$3 95
SW-2*	**CHECKBOOK** record	WB-2	**PET String and Array Handling**	$3 95
SW-3*	**ACCOUNTS** keep track of who owes you how much	WB-3	**PET Graphics**	$4 95
SW-4	**MEDIT** create and maintain data files	WB-4	**PET Cassette I/O**	$4 95
SW-5*	**CALENDAR** appointments, meetings at a glance	WB-5	**Miscellaneous PET Features**	$3 95
		WB-6	**PET Control and Logic**	$3 95

*These programs are special purpose data base management systems. They all can
- Sort numeric or string fields
- Select based on numeric or string (=)
- Select based on substring match
- Select based on range of entry number

Prices: **$9.90 each for programs using cassette storage for data**
$12.95 each using sequential floppy disk storage for data.
$5.95 each for instruction manual, approximately 40 pages.

Add $1.50 for shipping and handling

Money back guarantee

TIS
P.O. Box 921, Dept. PP
Los Alamos, NM 87544

PET is a trademark of Commodore Business Machines

• PET • PET • PET • PET • PET • PET • PET • PET • PET • PET • PET • PET • PET • PET

By taking the integer value the round off error is removed form the left half. Using STR$ takes advantage of the fact that the PET prints '5' even though it is not exactly 5; this takes care of the round off error on the right side. Comparing the numbers as strings removes the 'invisible' round off error.

Problems also exist with the trig functions. Try this:

```
    PRINT COS(0):IF COS(0)=1 THEN PRINT "OK"
```

Even though it prints1 it really isn't. Again STR$ will do the trick:

```
    IF STR$(COS(0))="1" THEN PRINT"OK"
```

Comparing numbers as strings should take care of the PET's round off error problems.

LIPS Meeting Information

This newsletter will carry announcements of LIPS meetings. Since editing, printing and mailing the newsletter takes some time, announcments may not reach everyone on time. Please mark the following dates on your calendar for LIPS meetings.

December 11 January 8 February 12

All meetings are on Thursdays at 4:00 PM at Harborfields HS, Greenlawn. Directions and more meeting information may be obtained by calling (516) 585-2402 between 7:00 and 10:00 PM or (516) 261-4900 ex. 191 between 8:00 AM and 2:00 PM (ask for Ralph Bressler). Any interested persons are invited to attend. Please watch for changes in each issue of The PAPER.

The Harborfields Computer Center has 1 8K old ROM PET, 9 8K new ROM PETs and 2 32K new ROM PETs. We also have a 2040 Dual Floppy Drive and a 2022 Tractor Feed Printer. All you really have to bring to the meetings is unique hardware to show off, software to trade (not copyrighted), a medium for copying programs and some friends.

The PAPER
Box 524
East Setauket, NY 11733